Limit Point Systems, Inc.

# Introduction

This document shows how to use the basic features of the SheafSystem. It is intended as an introduction for beginners, providing a guided tour of the essential capabilities of the SheafSystem libraries. It does not show how to use every feature of the system; readers are expected to use it as a starting place for further exploration of the reference documentation and class libraries.

The C++ examples in the document are available as source code in the examples subdirectory of the SheafSystemProgrammersGuide module and the reader is encouraged to build and execute the examples along with reading the text. The examples are numbered and the source for example N is in exampleN.cc. There are a few compilation and execution examples in the text, these are given in Linux using the csh shell, Gnu C++, and Gnu make.

# What you'll need

To take full advantage of this document, you'll need a few things in addition to the document itself, namely:

* an installed copy of the SheafSystemProgrammersGuide module, which includes the examples,
* a C++ compiler,
* a web browser, for viewing the reference documentation, and
* an installed copy of the SheafSystem.

# The SheafSystem installation

The SheafSystem installer installs all the files of the SheafSystem in a directory tree. We will have to refer to the root of this directory tree repeatedly, so to simplify the notation, we'll let <sheaf\_dir> refer to the full path to the root directory of the installation, for instance:

<sheaf\_dir> = /usr/local/SheafSystem

Wherever you see <sheaf\_dir> in this document, mentally replace it with the full path to your SheafSystem installation.

The installation includes 4 configurations of the libraries: Debug-contracts, Debug-no-contracts, Release-contracts, and Release-no-contracts. The "Debug" configurations are unoptimized and contain symbol information for use by interactive debuggers such as gdb. The "Release" configurations are optimized and contain no debugging information. We'll describe "Contracts" below. Generally speaking, the Release configurations are higher performance that the Debug configurations and the no-contracts configurations are much faster than the contract configurations.

The examples will compile and execute with any configuration, but we will always use the Debug-contracts configuration in the text below.

# Part I: The sheaf component

## Getting started

### PartSpace metaphor

The Part Space document describes the fundamental concepts of the SheafSystem in non-mathematical terms using the common notion of basic and composite parts, tables, and table schema. This document assumes the reader is familiar with the Part Space metaphor.

### Sheaf tables

As described in the Part Space document, a SheafSystem database is a collection of tables. Each table is equipped with a covering relation graph describing the lattice order of its rows and another graph describing the lattice order of its columns. Each such object table has an associated table called its schema table and the row graph of the schema table defines the column graph of the object table. A member of the row lattice is represented by a node in the row graph. A member also has a corresponding row in the table if and only if it is a basic part, a join irreducible member (jim) in the row lattice.

There are 3 special tables. the primitive schema table, the primitives table, and the namespace table. The primitives schema table terminates the schema recursion, it is its own schema table. The primitives table describes each primitive type supported by the system.

### Namespaces

A namespace table is a special table in each database that serves as a container and table of contents for all the other tables. The SheafSystem includes 3 predefined namespace types: the sheaves\_namespace, the fiber\_bundles\_namespace, and the geometry\_namespace. Each of these predefines the sheaf schema for the C++ types defined in the sheaf, fiber\_bundle, and geometry components, respectively. (The fields component doesn't have its own schema).

Creating an instance of a namespace is typically the first thing a client must do to use the SheafSystem, so we start with an example of how to do that using the most basic namespace, sheaves\_namespace. This example will also cover the basic mechanics of compiling and linking with the SheafSystem.

#### Example 1: Hello, Sheaf

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

// Create a standard sheaves namespace.

sheaves\_namespace lns("Hello-sheaf");

// Write its name to cout.

cout << lns.name();

return 0;

}

This code is in the SheafSystemProgrammersGuide module in examples/sheaf/example1.cc along with a Makefile:

#

# Full path to your C++ compiler, for instance /usr/bin/g++

#

CXX = /usr/bin/g++

#

# Full path the SheafSystem installation include and library directories

#

SHEAF\_INC\_DIR =<sheaf\_dir>/include

SHEAF\_LIB\_DIR = <sheaf\_dir>/Debug-contracts/lib

example1: example1.cc

$(CXX) -o example1 -I$(SHEAF\_INC\_DIR) -L$(SHEAF\_LIB\_DIR) example1.cc -lsheaves

To compile and link the example, you first have to configure the Makefile to your installation by setting the 3 variables CXX, SHEAF\_INC\_DIR, and SHEAF\_LIB\_DIR to the actual values for your installation. Then we can compile and link by:

>make example1

This command will compile example1.cc and link it with the shared library libsheaves.so to create an executable example1 in the current directory. We have to tell the dynamic loader where to find the shared library by setting the environment variable LD\_LIBRARY\_PATH to contain the path to the SheafSystem library directory, that is, the same value we set SHEAF\_LIB\_DIR to in the Makefile, for instance:

>setenv LD\_LIBRARY\_PATH <sheaf\_dir>/Debug-contracts/lib

Now we can execute the example:

>./example1

Hello-sheaf

That's the basic mechanics of creating an application with the SheafSystem. We've created a sheaves\_namespace in this example, but before we can do much with it, we need to learn a few programming patterns that the SheafSystem uses repeatedly.

## Programming patterns

There are a few design features shared by all the classes in the SheafSystem. In this section we will give a quick introduction to the most ubiquitous of these patterns. We'll introduce some more patterns later, as we need them, and also go into some of these initial patterns in more detail.

### Design by contract

The sheaf system is implemented using the "design by contract" programming paradigm. We'll cover the essentials of the method and how they are used in the SheafSystem. For a more detailed introduction, see the excellent book Design By Contract, by Example by Richard Mitchell and Jim McKim.

When using design by contract, each class is equipped with an invariant, a set of assertions that must be true at any time control returns the client. (The invariant is not defined when control is within a member function of the class.) Every member function is equipped with preconditions and postconditions. The preconditions are assertions that must be true when control enters the member function; the postconditions must be true when control leaves the member function. The "contract" in "design by contract" is between the client and the member function: if the client guarantees the preconditions are true, the member function ensures the invariant and the postconditions are true.

The invariant, precondition, and postcondition assertions are specified using "invariance", "require", and "ensure" macros, respectively, in the source code. If contracts are enabled when the library is compiled, these clauses will be evaluated as part of the execution of the member functions. If the conditions specified in the clauses are not true, execution throws an exception with an error message, which usually terminates the program.

The contracts are extremely useful for detecting improper use of the classes and member functions and are thus an important debugging tool. Once client code is correct, the contracts can be disabled to improve efficiency.

The SheafSystem Debug-contracts and release-contracts configurations are compiled with contracts enabled. The Debug-no-contracts and Release-no-contracts are compiled with contracts disabled.

The contracts are also published as an essential part of the reference documentation and are critical to using the sheaf system correctly. Let's look at the reference documentation for the sheaves\_namespace constructor we used in example1. The reference documentation is generated in html, so you can open it with your browser. The main page is <sheaf\_dir>/documentation/C++/index.html. If you browse to the documentation for class sheaves\_namespace and click on the constructor sheaves\_namespace(const string& xname), you'll find:

sheaf::sheaves\_namespace::sheaves\_namespace ( const string & *xname* )

Creates a sheaves namespace with name xname.

Precondition

* poset\_path::is\_valid\_name(xname)

Postcondition

* [invariant()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#a952742bdad45c56c22fd9509a00e9c07)
* [name()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#aec09bcd260a52a459c8a35ae5bc1bef5) == xname
* !in\_jim\_edit\_mode()
* [host()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#adc8f6d6d2b952a6842a1d09de75bff9a) == 0
* !index().[is\_valid()](http://192.168.4.199/comp-tutorial-dev-4/df/d4b/namespacesheaf.html#a3dd8f96a360e1b63c6caa744e5ccf7b3)
* [index()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#a9a283b1819bc8e75b212bff26fc645b0).same\_scope(member\_hub\_id\_space(false))
* [has\_standard\_subposet\_ct()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#af5786ce90013ec6e72dbacd9b67e1c13)
* [current\_namespace()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#ae8ca3a11bc745cf0b275a70ab71b2d70) == this
* [state\_is\_not\_read\_accessible()](http://192.168.4.199/comp-tutorial-dev-4/d1/d3c/classsheaf_1_1read__write__monitor__handle.html#adc32a6090b2df1e5673444d5170539f3)

So what does this tell us? The precondition:

* poset\_path::is\_valid\_name(xname)

tells us exactly what conditions the argument xname has to satisfy if we want this call to the constructor to work correctly, namely is\_valid\_name(xname) has to be true. Well, what does that take? If we look up poset\_path∷is\_valid\_name we find:

static bool sheaf::poset\_path::is\_valid\_name( const string &  xname )

True if xname is not empty and contains only name legal characters.

Postcondition

* result == (!xname.[empty()](http://192.168.4.199/comp-tutorial-dev-4/d0/d38/classsheaf_1_1poset__path.html#a05ee8f14bcc22701b551059341f16749) && (xname.find\_first\_not\_of([name\_legal\_characters()](http://192.168.4.199/comp-tutorial-dev-4/d0/d38/classsheaf_1_1poset__path.html#a34019af3a5bee6f34d3ec2c2657a8671)) == string::npos))

So xname can't be empty and can't contain any characters not in name\_legal\_characters(). If we click on name\_legal\_characters we find:

static const string & sheaf::poset\_path::name\_legal\_characters( )

The characters a name is allowed to contain.

Postcondition

* result == "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789\_ -,.=+()\*:?"

So xname has to be non-empty and contain only the above characters.

If xname satisfies these conditions, which it does in example1, then the postcondition gives a great deal of information about what state the sheaves\_namespace object is in after construction.

The first postcondition is:

* [invariant()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#a952742bdad45c56c22fd9509a00e9c07)

that is, the invariant has to be satisfied. As we said above, this is an implicit postcondition of every member function, even if we don't explicitly provide it as part of the contract. So what does this mean for sheaves\_namespace? Well, click on invariant() to find:

virtual bool sheaf::namespace\_poset::invariant ( ) const

Class invariant.

Invariant

* poset\_state\_handle::invariant()
* host() == 0
* !index().is\_valid()
* !is\_external()
* is\_attached() ? primitives().is\_attached() : true
* is\_attached() ? (primitives().index() == PRIMITIVES\_INDEX) : true
* state\_is\_read\_accessible() ? primitives().state\_is\_read\_accessible() : true
* is\_attached() ? primitives\_schema().is\_attached() : true
* is\_attached() ? (primitives\_schema().index() == PRIMITIVES\_SCHEMA\_INDEX) : true
* state\_is\_read\_accessible() ? primitives\_schema().state\_is\_read\_accessible() : true

Sheaves\_namespace inherits namespace\_poset and doesn't override the invariant, which is a virtual function, so the invariant of sheaves\_namespace is the invariant of namespace\_poset. The invariant in a derived class must be at least as strong as the invariant in the base space, so the invariant of namespace\_poset calls the invariant of its base class, poset\_state\_handle. Beyond whatever poset\_state\_handle∷invariant() ensures, the namespace\_poset invariant ensures several properties of the data members, primitives() and primitives\_schema() in particular.

As this invariant shows, the conditional expression

* x ? y : true

appears frequently in the contracts, so it is worth describing in more detail. As an assertion, this expression can be read "x implies y", that is, x can be either true or false, but if x is true, then y must be true as well. If x is false, there is no condition on y.

The reader is encouraged to examine the poset\_state\_handle invariant to learn what additional invariances sheaves\_namespace has inherited, but we'll move on to the rest of the postcondition of the constructor. The next postcondition is one you'd expect:

* [name()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#aec09bcd260a52a459c8a35ae5bc1bef5) == xname

that is, the name of the namespace is the name we gave it.

The remainder of the postconditions ensure various arcane properties of the namespace that we're not very interested in right now. But when your tackling a tough debugging problem, any of these may be just the piece of information you need!

The power of the design by contract method comes from the great amount of detailed information contained in the assertions and two further properties. First, if contracts are turned on, that is if you are using either the Debug-contracts or Release-contracts configuration of the library, the pre- and post-conditions of a function are executed whenever the function is called. Second the contracts exhibited in the documentation are extracted directly from the source code. The combination of the two allows you to reason about the behavior of the code with great confidence while designing, programming, and especially while debugging.

So what happens if the contract for a member function is not satisfied? Let's find out by trying to create a sheaves\_namespace without a name.

#### Example 2: contract for sheaves\_namespace constructor.

#include "sheaves\_namespace.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example2:" << endl;

// Attempt to create a standard sheaves namespace

// with an empty name. This violates the preconditions

// of the constructor and will throw an exception and abort.

sheaves\_namespace lns("");

// Done.

return 0;

}

If we compile and run this, assuming we still have LD\_LIBRARY\_PATH set from running example1, we get:

>make example2

>./example2

terminate called after throwing an instance of 'std::logic\_error'

what(): 'poset\_path::is\_valid\_name(xname)' in file namespace\_poset.cc at line 1941

Abort

The error message tells you exactly what assertion failed. If you're debugging, you can walk back up the stack from where the exception was actually thrown to the assertion that failed and inspect local variables, for instance xname, to determine what went wrong.

### Concurrency control

One of the attractive features of the sheaf data model is that its mathematical formalism provides a natural language for describing concurrency and parallelism. The sheaf system libraries were designed for concurrent programming using an access control mechanism based on the monitor design pattern. Currently, this mechanism is only partially implemented and the SheafSystem libraries are delivered with the access control mechanism disabled. Programmers nevertheless must be aware of certain aspects of the access control mechanism, which we describe in this section. More complete examples are included in Appendix A.

Access to every table is controlled. A client thread can have no access, read access, or read-write access. At any given time, either no client has access, exactly one client has read-write access, or one or more clients have read access. Before reading or writing a table or any of its members, a client must request read access or read-write access, respectively. After accessing the table, the client must release access. If a client requests read access and another client already has write access, or vice versa, the request blocks until the other client releases the conflicting access.

The concurrency control mechanism is "enforced" through precondition clauses in the table member functions. In order to make concurrency control apparent to the client and avoid dead lock, the library routines do not themselves request or release access without the client knowing it. Instead, they "publish" their access requirements as preconditions and let the client control the access.

For instance, in example 1 we invoked sheaves\_namespace∷name(). Consulting the reference documentation, we find for the name() member function:

virtual string sheaf::namespace\_poset::name() const

The name of this namespace.

Precondition

* state\_is\_read\_accessible()

So, if the access control mechanism is enabled, the client must request read access, invoke the name function, and release access:

lns.get\_read\_access();

cout << lns.name() << endl;

lns.release\_access();

Getting and releasing access can be a tedious programming chore. Furthermore, it is syntactically impossible in some cases, for instance within a pre- or post-condition clause. So many member functions offer an "auto-access" option. These routines will automatically get and release the access they need, if the client allows it by setting an auto-access argument to true. If invoked with the auto-access argument false, the client must get the required access before making the call. These routines also publish their access requirements as preconditions. For instance, the auto-access version of the name function is:

virtual string sheaf::namespace\_poset::name( bool  xauto\_access ) const

The name of this namespace.

Precondition

* state\_is\_auto\_read\_accessible(xauto\_access)

Using this version of the name function, the client need only invoke the function with argument "true":

cout << lns->name(true) << endl;

The function will request read access, get the name, release access, and return the name.

When the access control mechanism is disabled, the client always has read-write access and neither requesting nor releasing access is necessary. Functions with an auto-access argument can be called with either true or false, either will work. However, the access control mechanism doesn't quite disappear from the programmer's view. The auto-access signatures are still present and the access requirements still appear as preconditions in the contracts.

### Handles and states

The Sheaf System is object-oriented, so the client interacts with the library by manipulating the various objects presented by the library interface. Lattice members are a prime example. Many of the objects exported by the interface are not however stored as explicit objects internally. Both memory and performance efficiency often require that such objects be implicit - stored as disjoint data items in bulk arrays. The problem of how to present an externally explicit object interface to an internally implicit object is a common software design problem and several similar design patterns - flyweight, proxy, surrogate, etc., have been developed to address this problem. In the Sheaf System, we call such a surrogate object a handle and the internal data it accesses is called its state.

For the most part, the distinction between handles and states is an implementation detail that the client needs to be only vaguely aware of. The client uses the handle object as if it were stored internally without worrying about the internal details. But there is one aspect the client has to be aware of: the client has to somehow get a handle to the desired object and when finished with it the client may have to explicitly release it.

There are two basic patterns. In the first pattern, some object has a data member which is a handle and it provides an accessor to this data member. For instance, sheaves\_namespace, like every lattice, has a top member. This member is represented by a data member which is a handle and sheaves\_namespace exports an accessor:

sheaves\_namespace lns;

namespace\_poset\_member& ltop = lns.top()

The namespace object allocated and owns the handle. The client need not and should not worry about releasing or otherwise deallocating the handle.

The second pattern addresses the more general case in which the number of handles the client needs and what states they should be attached to is not known at compile time. In order to support efficient allocation and deallocation of handles, the system maintains pools of handles which the client can "borrow", use, and return. For instance, we'll see in the next section that index spaces are accessed via handles and the client can get a handle from the appropriate index space family:

index\_space\_handle& lids =

lns.member\_id\_spaces(true).get\_handle("member\_poset\_id\_space");

When accessed in this way, the handle must be released when the client is finished with it:

lns.member\_id\_spaces(true).release\_handle(lids, true);

How does a client know whether to release a handle or not? Simple, if you got the handle by calling get\_<whatever>, you need to release it by calling release\_<whatever>. Release if and only if get!



Figure 1: Hub and spoke architecture of an index space family.

## Index spaces and scoped indices, part 1

The members of the row lattice of a table (and hence the members in the column lattice as well) are identified by integer ids. Subsets of the members are very important in the SheafSystem and it is frequently useful to generate a special purpose index scheme for a given subset. Such an index scheme is referred to as an "index space", or "id space" for short. The SheafSystem provides extensive support for defining and using id spaces.

### Index spaces and iterators.

More specifically, an index space is a set of integer ids. The system supports the creation and use of a family of index spaces. The fundamental id space of the family is the member id space - the ids automatically generated for the nodes in the row graph. This index space is called the hub id space because the index space family has a hub and spoke architecture as shown in Figure 1. As you can see from the diagram, there are several different kinds of id space and even two hub id spaces, the "unglued" and "glued" versions. For a detailed discussion of this structure see the document "Index Spaces". For the moment, hub id space means unglued hub id space and you should just think of each id space on the rim as a way of indexing some subset of the hub id space, with each spoke representing a map. We'll focus on the basics of how to use id spaces.

As one might expect, the principal use for a member id is to access the features of the member the id refers to. The principal use of a member id space is to iterate over all the members in the subset defined by the id space. Let's look at an example.

### Example 3: Iterates over the member hub id space.

#include "hub\_index\_space\_handle.h"

#include "index\_space\_iterator.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example3:" << endl;

// Create a standard sheaves namespace.

sheaves\_namespace lns("Example3");

// Get a handle for the member hub id space.

const index\_space\_handle& lmbr\_ids = lns.member\_hub\_id\_space(true);

// Find out how many ids are in the id space.

cout << lmbr\_ids.name();

cout << " has " << lmbr\_ids.ct() << " ids.";

cout << endl;

// Id spaces are defined as half open intervals, like STL iterators.

// If the space is "gathered", begin() == 0 and end() = ct().

// If the space is not gathered, it's "scattered".

cout << "beginning at " << lmbr\_ids.begin();

cout << " and ending at " << lmbr\_ids.end();

cout << " " << (lmbr\_ids.is\_gathered() ? "gathered" : "scattered");

cout << endl;

// The main thing one does with id spaces is iterate over them.

// Get an iterator from the iterator pool.

index\_space\_iterator& lmbr\_itr = lmbr\_ids.get\_iterator();

cout << endl << "Iterate:" << endl;

while(!lmbr\_itr.is\_done())

{

// The current member of the iteration is "pod()".

// "POD" is an ISO C++ acronym for "plain old data".

// A pod is an ordinary integer id, in contrast with

// a "scoped\_index" id, to be discussed shortly.

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

// Use the id to get the member name.

// Member name requires a hub id, but since we're using

// the hub id space, pod and hub pod are the same thing.

cout << "id: " << lpod;

cout << " hub id: " << lmbr\_itr.hub\_pod();

cout << " name: " << lns.member\_name(lpod, true);

cout << (lns.is\_jim(lpod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// You can reuse an iterator by resetting it.

lmbr\_itr.reset();

cout << endl << "Reiterate:" << endl;

while(!lmbr\_itr.is\_done())

{

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

cout << "id: " << lpod;

cout << " hub id: " << lmbr\_itr.hub\_pod();

cout << " name: " << lns.member\_name(lpod, true);

cout << (lns.is\_jim(lpod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// If you got an id space or iterator from the pool with get\_

// you have to return it to the pool with release\_.

lmbr\_ids.release\_iterator(lmbr\_itr);

// The id space itself is a data member of the id space family,

// we didn't get it from the pool with get\_, so we don't have to

// release it.

// Exit:

return 0;

}

If we execute example3 we get:

>./example3

SheafSystemProgrammersGuide Example3:

\_\_hub has 6 ids.

beginning at 0 and ending at 6 gathered

Iterate:

id: 0 hub id: 0 name: bottom is a jrm.

id: 1 hub id: 1 name: top is a jrm.

id: 2 hub id: 2 name: primitives\_schema is a jim.

id: 3 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 4 hub id: 4 name: primitives is a jim.

id: 5 hub id: 5 name: schema definitions is a jrm.

Reiterate:

id: 0 hub id: 0 name: bottom is a jrm.

id: 1 hub id: 1 name: top is a jrm.

id: 2 hub id: 2 name: primitives\_schema is a jim.

id: 3 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 4 hub id: 4 name: primitives is a jim.

id: 5 hub id: 5 name: schema definitions is a jrm.

### Id maps and scoped ids.

As we said above, id spaces are used for indexing subsets. For instance, in a namespace, the member poset id space indexes just the jims, which represent the member posets - the other posets contained in the namespace. There may be several or even many id spaces available in a practical setting. Various member functions may require an index to be in a particular id space, most commonly in the hub id space. The id maps associated with the spokes in the id space family provide the mechanism for translating between id spaces.

Every id space has a map to the (unglued) hub id space. The index\_space\_handle class provides member functions for mapping ids between the id space and the hub id space:

pod\_type hub\_pod (pod\_type xid) const

The pod index in the unglued hub id space equivalent to xid in this id space; synonym for unglued\_hub\_pod(pod\_type).

and

pod\_type pod (pod\_type xid) const

The pod index in this space equivalent to xid in the hub id space.

Using these functions we can map between id spaces. For instance, if id1 is an id in id\_space1 and id\_space2 is a different id space, then

pod\_type id2\_eqv\_1 = id\_space2.pod(id\_space1.hub\_pod(id1));

is the id in id\_space2 that identifies the same member identified by id1 in id\_space1, if such an equivalent member exists. The postcondition for the pod(pod\_type) function is:

* !is\_valid(result) || contains(result)

So if id\_space2 does not have an equivalent member, id2\_eqv\_1 is assigned an invalid value. The value

sheaf::pod\_index\_type sheaf::invalid\_pod\_index()

The invalid pod index value.

is reserved as a "null" value for index types. It is currently set to numeric\_limits<pod\_index\_type>::max(), but that may change and I only mention that so you will recognize it if you see it in a print out or in the debugger. It should be used as an opaque value. A pod type can be tested for validity using:

bool sheaf::is\_valid(sheaf::pod\_index\_type xpod\_index)

True if an only if xpod\_index is valid.

Using the pod and hub\_pod functions, the programmer can map ids between id spaces. But it can be tedious. Worse, it may be difficult or impossible for a programmer to track just what id space a given index is in. The scoped\_index class provides a convenient mechanism for both managing the connection between an id and the space it belongs to and for automatically mapping between id spaces. We call the id space an id belongs to the scope of the id. A scoped\_index is a pair (id, scope). Most member functions that require an id as input are available in two signatures; one signature that takes a pod id and one that takes a scoped id. One can use a scoped id, once it has been initialized, without worrying what scope it is in; any function that accepts a scoped id will translate it to the scope it requires. We'll see more complex examples of mapping between id spaces later, for now let's redo example3 using the member poset id space, the hub id space, id maps and scoped ids.

### Example 4: Iterates over the member poset id space.

#include "index\_space\_handle.h"

#include "index\_space\_iterator.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example4:" << endl;

sheaves\_namespace lns("Example4");

// Get a handle for the member poset id space;

// has one member for each poset in the namespace.

const index\_space\_handle& lmbr\_ids =

lns.get\_member\_poset\_id\_space(true);

// Print out the same info we did for the hub id space.

cout << lmbr\_ids.name();

cout << " has " << lmbr\_ids.ct() << " ids.";

cout << endl;

cout << "beginning at " << lmbr\_ids.begin();

cout << " and ending at " << lmbr\_ids.end();

cout << " " << (lmbr\_ids.is\_gathered() ? "gathered" : "scattered");

cout << endl;

index\_space\_iterator& lmbr\_itr = lmbr\_ids.get\_iterator();

cout << endl << "Iterate:" << endl;

while(!lmbr\_itr.is\_done())

{

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

// Use the id to get the member name.

// Member name requires a hub id which we can get in two ways.

// The id space will use the map from the id space to the hub

// to translate any id in the id space to its equivalent in the

// hub:

index\_space\_iterator::pod\_type lhub\_pod = lmbr\_ids.hub\_pod(lpod);

// The iterator can provide the hub id equivalent for the current

// id, and it can be faster because for some id space types it can

// avoid the map lookup.

lhub\_pod = lmbr\_itr.hub\_pod();

cout << "id: " << lpod;

cout << " hub id: " << lhub\_pod;

cout << " name: " << lns.member\_name(lhub\_pod, true);

cout << (lns.is\_jim(lhub\_pod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// Most member functions are available with two signatures, one

// that takes a pod\_index\_type and one that takes a scoped\_index.

// If you don't want to think about what the scope for an argument

// should be, you can use the scoped\_index signature.

// Create a scoped id with scope = member poset id space.

scoped\_index lscoped\_id(lmbr\_ids);

// The value sheaf::invalid\_pod\_index() is reserved as a

// "null" value for index types. It is currently set to

// numeric\_limits<pod\_index\_type>::max(), but don't count on it.

cout << endl << "sheaf::invalid\_pod\_index()= ";

cout << sheaf::invalid\_pod\_index() << endl;

// When a scoped id is created without a specific pod value,

// it is invalid by default.

cout << "lscoped\_id= " << lscoped\_id;

cout << " is\_valid() ";

cout << boolalpha << lscoped\_id.is\_valid() << noboolalpha;

cout << endl;

// Reset the iterator and re-iterate using

// the scoped\_index signature for member\_name.

lmbr\_itr.reset();

cout << endl << "Reiterate:" << endl;

while(!lmbr\_itr.is\_done())

{

// Set the scoped id for the current member of the iteration.

lscoped\_id.put\_pod(lmbr\_itr.pod());

// Assignment is overloaded, so you can also say:

lscoped\_id = lmbr\_itr.pod();

// Use the scoped\_index signature to get the member name.

cout << "scoped\_id: " << lscoped\_id;

cout << " name: " << lns.member\_name(lscoped\_id, true);

cout << (lns.is\_jim(lscoped\_id) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

lmbr\_ids.release\_iterator(lmbr\_itr);

// Exit:

return 0;

}

When we run example4 we get:

>./example4

SheafSystemProgrammersGuide Example4:

member\_poset\_id\_space has 3 ids.

beginning at 0 and ending at 3 gathered

Iterate:

id: 0 hub id: 2 name: primitives\_schema is a jim.

id: 1 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 2 hub id: 4 name: primitives is a jim.

sheaf::invalid\_pod\_index()= 2147483647

lscoped\_id= (2, 2147483647) is\_valid() false

Reiterate:

scoped\_id: (2, 0) name: primitives\_schema is a jim.

scoped\_id: (2, 1) name: namespace\_poset\_schema is a jim.

scoped\_id: (2, 2) name: primitives is a jim.

Note the value of the invalid id in the above output; we'll see it again shortly.

## Storage\_agent

We've talked about the notion of a SheafSystem database, so there must be some way to make a namespace persistent and indeed there is. Persistent storage is managed by the storage\_agent class. A storage\_agent makes it particularly easy to save an entire namespace to disk, as we show in the next example.

### Example 5: Write a namespace to a file

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example5:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example5");

// Write the namespace to a file.

storage\_agent lsa("example5.hdf");

lsa.write\_entire(lns);

// Exit:

return 0;

}

If we build and run example5, it writes the file named in the storage\_agent constructor, "example5.hdf".

>./example5

>ls \*.hdf

example5.hdf

We'll see shortly how we can view the contents of this file.

## Viewing Namespaces

Once we have a namespace, we'd like to know what it contains. We've already seen how to iterate over the members of the namespace and display their names. Now we'll look at 3 ways that are easier and provide a lot more information.

### Stream insertion operator

The base class for namespaces, namespace\_poset, has a stream insertion operator for writing the contents of the namespace to a stream. The insertion operator is most commonly used for dumping a namespace to cout for debugging purposes.

#### Example 6: Write namespace to cout

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example6:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example6");

// Write the namespace to cout.

cout << lns << endl;

// Exit:

return 0;

}

When we execute example6, it creates quite a lot of output, even for sheaves\_namespace, which is as close to being empty as a namespace can get. We won't include it here, but the reader should examine the file example6.cout in the same directory with example6.cc.

For each poset in the namespace, including the namespace itself, the stream insertion operator prints information about the row graph, the subposets of the row graph, and the table. We'll learn more about how to interpret all this information as we go along.

### The dump\_shf utility

The SheafSystem provides the dump\_shf utility for reading a sheaf file and dumping its contents to cout using the stream insertion operator. So if you've written a sheaf file, as we did in example 5, then we can view its contents easily.

#### Example 7: View namespace with dump\_sheaf

First, make sure you have set your environment using the set\_env\_vars script in the build directory of your Programmer's Guide installation:

>source set\_env\_vars.csh

Then run the dumpsheaf utility in the bin directory of the SheafSystem installation:

<sheaf\_dir>/Debug\_contracts/bin/dumpsheaf example5.hdf

### The SheafScope interactive file browser

The SheafScope is another SheafSystem utility. It provides an interactive, graphical browser for sheaf files.

#### Example 8: View namespace with SheafScope

Make sure you've set your envornoment with the set\_env\_vars script as shown above, then run the SheafScope.

>java -jar SheafScope.jar example5

## Posets

### Table or part space or lattice or poset?

We've so far talked about a sheaf database being a collection of sheaf tables. In the Part Space tutorial we talked about sheaf tables as part spaces and in the Analysis and Design tutorial we revealed that a sheaf table could be thought as either a poset-ordered or lattice-ordered relation. So which is it? The answer of course, is (e) all of the above. But when it comes to naming classes, we had to pick one. The one we picked was poset. Most of the classes that implement sheaf tables are posets of some kind or another. The most commonly used type is class poset. The type that is used to represent meshes in the fiber bundles component is base\_space\_poset. The abstract base class for all poset types is poset\_state\_handle.

|  |
| --- |
| **Historical note**: Poset\_state\_handle, the abstract base class for all poset types, is as its name says, a handle. But this is a historical artifact. There is no longer any reason for it to be a handle, the state of a poset is an explicit object, and the various handle features are in fact protected so they can't be used. |

### Creating posets

A namespace is a factory for posets. Posets are created using the member function template new\_member\_poset<T>, where T is the type of poset to create:

T& new\_member\_poset (const string &xname,

const poset\_path &xschema\_path,

const arg\_list &xargs,

bool xauto\_access)

Creates a new poset with name xname, schema specified by xschema\_path, and table attributes initialized by xargs.

But first, we have to have a schema if we want to create a poset. Typically, this means we have to create a schema poset before creating an object poset. We'll see how to create a schema poset shortly, but we can avoid creating a schema poset if the poset we want to create has only a single attribute. Instead, we can use a member of the primitives poset that every namespace has as a schema. We'll do that to get started.

#### Poset\_path

The schema for a poset is a member of a schema poset and a member can be identified by a path. A poset path is similar to a file path, but has only two elements, a poset name and a member name. We can create a path by specifying the poset name and member name separately:

poset\_path lschema\_path("primitives", "INT");

or by specifying a complete path, like a file path:

poset\_path lschema\_path("primitives\_poset/INT");

#### Arg\_list

Some types of poset cannot be default constructed, so the factory method requires an argument list for the constructor. Each poset type provides a static function make\_args(...) that sets up the arg\_list. Every poset type has such a function, even if it can be default constructed. In that case, the make\_args function itself has no arguments and the arg\_list it creates is empty. This is in fact the case for the ordinary poset class we want to construct:

arg\_list largs = poset::make\_args();

The arg\_list class has a constructor that takes a string, so if you know the arg\_list is empty can also just pass an empty string for the arg\_list, but we'll do it the general way for now.

So now all we need is a name and we can construct the poset. We'll call it "simple\_poset":

poset& lposet = lns.new\_member\_poset<poset>("simple\_poset",

lschema\_path, largs, true);

### Accessing posets

Once you've created a poset and have a reference to it, you can access its features. For instance, you can find out what its id is:

cout << lposet.index().hub\_pod() << endl;

All poset types have a stream insertion operator, so once you have access to a poset, you can use the stream insertion operator to print it out:

cout << lposet << endl;

We'll do more with posets, like creating members, in the next section.

In the meantime, what if a poset already exists, how do you get a reference to it? You can get a reference to a poset by id, which is available in two variants, pod and scoped\_index:

poset\_state\_handle & member\_poset (pod\_index\_type xindex, bool xauto\_access) const

The poset\_state\_handle object referred to by the member with hub id xindex.

poset\_state\_handle & member\_poset(const scoped\_index &xindex, bool xauto\_access) const

The poset\_state\_handle object referred to by the member with index xindex.

For instance, assuming that the id of the poset we created above is 5, we can get a reference to it with:

poset\_state\_handle& lpsh1 = lns.member\_poset(5, true);

You can also access it by path. The member name part of the path can be empty or not, only the poset name will be used.

poset\_state\_handle & member\_poset (const poset\_path &xpath, bool xauto\_access) const

The poset\_state\_handle object referred to by the member with name xpath.poset\_name().

Furthermore, since poset\_path has a constructor that takes a string literal, you can specify the path as a string literal:

poset\_state\_handle& lpsh2 = lns.member\_poset("simple\_poset", true);

Poset\_state\_handle is the abstract base class for all poset types. If you know the specific type of a poset and want a reference to that type, all three of the above signatures are also available in a templated version:

P& member\_poset(pod\_index\_type xindex, bool xauto\_access) const;

The poset\_state\_handle object referred to by the member with hub id xindex dynamically cast to type P&.

invoked with bracket notation:

poset& lposet1 = lns.member\_poset<poset>("primitives", true);

### Deleting posets

Posets live in the namespace. Once you've created one, it stays in the namespace whether you have a reference to it or not. If you want to delete a poset, use the namespace delete\_poset function, available in the same id and path signatures as member\_poset, for instance:

void delete\_poset (const poset\_path &xpath, bool xauto\_access)

Delete the poset with name xpath.poset\_name().

Example 10 ties the namespace functions we've been discussing into a single example.

### Example 9: Creating, accessing, and deleting posets.

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example9:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example9");

// We use a schema with a single integer attribute.

poset\_path lschema\_path("primitives", "INT");

// The constructor for the ordinary poset class

// doesn't need any arguments.

arg\_list largs = poset::make\_args();

// Create the poset.

poset& lposet = lns.new\_member\_poset<poset>("simple\_poset",

lschema\_path, largs, true);

// Print the poset to cout.

cout << lposet << endl;

// Write the namespace to a sheaf file.

storage\_agent lsa("example9.hdf");

lsa.write\_entire(lns);

// You can get another reference to the poset by id,

// if you know the id:

poset\_state\_handle\* lpsh1 = lns.member\_poset(6, true);

// and by path:

poset\_state\_handle\* lpsh2 = lns.member\_poset("simple\_poset", true);

// Delete the poset by path.

// Invalidates all the above references.

lns.delete\_poset(lposet.path(), true);

// Exit:

return 0;

}

When we run example9, the output is once again a bit lengthy to include here, take a look at example9.cout.

## Poset members

We created a poset in the last section, but it was empty. Well, almost empty. As we discussed in the Part Space tutorial, a sheaf table represents a part space. A part space always has a bottom member// corresponding to the empty assembly of basic parts (jims). It also has a top member, corresponding to the assembly of all the basic parts. We automatically create these two composite parts (jrms) when we create the poset. When the poset has no basic parts, the top is equivalent to the bottom as an assembly, but they are still distinct members.

### Creating join irreducible members

We can't create any interesting jrms until we have some jims, so we'll start by creating some jims. A poset has a special editing mode, called "jim\_edit\_mode", for creating jims. Jrms can be created at any time, but to create a jim you have to put the poset into jim\_edit\_mode. Jim edit mode allows you to directly edit the row graph, creating or deleting members and cover links. Once the jims poset has been defined, you can create composite parts either directly by creating the members and links or algebraically, using the join and meet operations.

Jim edit mode is off by default, so we have enter jim edit mode to create a jim:

void begin\_jim\_edit\_mode (bool xauto\_access)

Allow editing of jims and jim covering relation.

(All the functions we will discuss in this section are member functions of class poset or its ancestors unless specifically scoped. Click the "List of all members" item in the extreme upper right of the poset class documentation web page to get a listing of all member functions, both direct and inherited.)

You create a jim with the new\_member function:

sheaf::pod\_index\_type new\_member (bool xis\_jim,

poset\_dof\_map \* xdof\_map = 0,

bool xcopy\_dof\_map = false )

Create a disconnected member with is\_jim == xis\_jim. If xdof\_map != 0, the new member uses it for dof storage, otherwise it creates an instance of array\_poset\_dof\_map. WARNING: this routine leaves a disconnected member in the poset and hence leaves the poset in an invalid state. The client must properly link the member created by this routine using new\_link in order to return the poset to a valid state.

|  |
| --- |
| **Historical note**: "Dof" is an acronym for "degree of freedom" but it means "attribute". A "dof\_map" is a tuple. Both are historical artifacts, originating in early attempts to interpret fields as relational tuples. |

To create a new jim, we call new\_member with xis\_jim true and let it create a tuple for the new member by accepting the default values for xdof\_map and xcopy\_dof\_map. Let's create three new jims, corresponding to the basic parts in the line segment example from the Part Space tutorial, Figure 10, which we reproduce in Figure 2. Well, almost reproduce, Figure 2 has a top member because sheaf tables always have top and bottom members.

The code to create the jims is:

lposet.begin\_jim\_edit\_mode(true);

pod\_index\_type lv0\_pod = lposet.new\_member(true);

pod\_index\_type lv1\_pod = lposet.new\_member(true);

pod\_index\_type ls0\_pod = lposet.new\_member(true);



Figure 2: Line segment example from Part Space, Figure 10.

### Ordering poset members

You define the ordering relation for the poset by explicitly creating cover links between the jims, using new\_link:

void new\_link( pod\_index\_type xgreater, pod\_index\_type xlesser)

Insert a cover link from greater to lesser (that is, xgreater covers xlesser). WARNING: this routine does not ensure that the link is a cover link, that is, it does not remove redundant or conflicting links. Improper use of this routine can produce inconsistent poset states.

Continuing with the line segment example, the segment member should cover the two vertices:

lposet.new\_link(ls0\_pod, lv0\_pod);

lposet.new\_link(ls0\_pod, lv1\_pod);

Each vertex is an atom, there is no smaller basic part than a vertex, so each vertex should have a cover link to bottom. Similarly, there is no larger part than the segment, so top should cover the segment. We can put these links in explicitly. The id of the top and bottom member are defined in the enumeration sheaf::standard\_member\_index as TOP\_INDEX and BOTTOM\_INDEX, respectively. Or, we can get the id from the top() or bottom() accessors:

lposet.new\_link(TOP\_INDEX, ls0\_pod);

lposet.new\_link(lv0\_pod, BOTTOM\_INDEX);

lposet.new\_link(lv1\_pod, lposet.bottom().index().pod());

However, we don't have to explicitly put the links to top and bottom. Remember that the cover relation graph is a directed graph, with links pointing in the "covers" direction. For a given member p, the set of lesser members p is linked to by outgoing links, that is the set of members p covers, is called the lower cover of p. The set of larger members that are linked to p by the incoming links, the set of members that cover p, is called the upper cover of p. It is an invariant of a lattice that bottom is the only member with an empty lower cover and top is the only member with an empty upper cover. The end\_jim\_edit\_mode function:

void end\_jim\_edit\_mode(bool xensure\_lattice\_invariant = true, bool xauto\_access = true)

Prevent editing of jims and jim order relation.

will enforce the invariant if we request it by setting the xensure\_lattice\_invariant argument to true. In that case, it automatically links anything with an empty lower cover to bottom and links top to anything with an empty upper cover. This requires a search of the graph for empty covers, so it is more efficient in large graphs to do the linking explicitly.

When we're finished creating and linking jims, we leave jim edit mode. We'll just take the default arguments, even though we've already linked everything, it won't make any difference for a tiny graph like this one:

lposet.end\_jim\_edit\_mode();

### Accessing poset members

We've created the basic parts and ordered them, but we haven't set any of their attributes, so let's do that now.

#### Member names

We've already seen that every poset member has at least one implicit attribute, it's id, automatically assigned by the system. Every member also has another implicit attribute, supported by the system but not automatically assigned: a name. Any member can be given a name, in fact any member can be given multiple names, but naming is optional. Names are assigned with the put\_member\_name function:

void put\_member\_name(pod\_index\_type xindex, const string & xname,

bool xunique, bool xauto\_access = false )

Make xname a name for the member with hub id xindex; if xunique, make xname the only name.

We'll give all our basic parts the obvious names:

lposet.put\_member\_name(lv0\_pod, "v0", true);

lposet.put\_member\_name(lv1\_pod, "v1", true);

lposet.put\_member\_name(ls0\_pod, "s0", true);

We can retrieve a name with the member\_name function:

string member\_name(pod\_index\_type xindex, bool xauto\_access = false)

A name for the member with hub id xindex.

For instance:

cout << lposet.member\_name(lv0\_pod);

Members can have more the one name. The reader is encouraged to review the other member name functions in the reference documentation for poset\_state\_handle.

#### Schema

The explicit attributes of a member are whatever is defined by the schema for the poset. The schema for a poset is a member of a schema poset. A handle for the schema member is available from the poset:

const schema\_poset\_member& schema () const

The schema for this poset (const version).

We'll learn more about member handles shortly. In the meantime, we'll just introduce a few features we need.

Mathematically, each attribute is a component of a tuple and components are traditionally accessed by component id. There is an id space defined for the attributes specified by each schema member. In fact, since the table is partitioned into a row part and a table part for storage efficiency, each schema member defines two id spaces, one for the row attributes and one for the table attributes. We can get either id space from the schema using the dof\_id\_space accessor function:

const index\_space\_handle&

schema\_poset\_member::dof\_id\_space(bool xis\_table\_dofs) const

The table dof (xis\_table\_dof true) or row dof id space for the schema defined by this.

For instance, the row attribute id space for our example is:

const index\_space\_handle& latt\_id\_space = poset.schema().dof\_id\_space(false);

Now, because we took a short cut defining the schema, we don't have all the attributes of the original example in the Part Space tutorial. All we have is a single integer attribute called "INT". Since we know that the schema has only a single row attribute, it has to be the first id in the id space, so we can get its id with:

pod\_index\_type latt\_pod = lposet.schema().dof\_id\_space(false).begin();

We can also create a scoped id for an attribute:

scoped\_index latt\_id(lposet.schema().dof\_id\_space(false), latt\_pod);

#### Member tuple

The attributes are components in the relation tuple of the member, so to access an attribute we need a reference to the tuple or "dof\_map", which we can get with:

poset\_dof\_map&

member\_dof\_map(pod\_index\_type xmbr\_index, bool xrequire\_write\_access)

The dof map associated with the member identified by xmbr\_index (mutable version).

We can get the attribute tuple for the first vertex for instance:

poset\_dof\_map& ltuple = lposet.member\_dof\_map(lv0\_pod, true);

#### Member attributes

Once we have the tuple, we can get an attribute value with the dof accessor function:

primitive\_value poset\_dof\_map::dof(pod\_index\_type xdof\_id) const

The dof with name xname.

and set an attribute value with the put\_dof mutator function:

void poset\_dof\_map::put\_dof(pod\_index\_type xdof\_id, const primitive\_value& xdof)

Sets the dof with name xname to xdof.

The accessor and mutator functions are each available in 3 signatures corresponding to specifying the attribute by pod id, scoped id, or name.

#### Primitive\_value

The attribute value returned by the accessor functions and accepted by the mutator functions is of type primitive\_value, which is essentially a wrapper for any primitive type. It allows a single signature for each function to support any attribute type. Primitive\_value is essentially a union that can store any primitive plus some type information. It also has conversion operators to and from every primitive type that make it easy to put a primitive value in to a primitive\_value and get it back again.

You can put a value into a primitive\_value with either a constructor or an assignment:

primitive\_value lpval(int(0));

lpval = float(1);

and get it back again with an assignment:

float lf = lpval;

You can find out what type of value a primitive\_value is currently holding with the id function by testing it against the id() member of the primitive\_traits template for the appropriate type:

if(lpval.id() == primitive\_traits<float>.id())

{

float lf = lpval;

}

#### Setting an attribute

Setting the attribute for the vertex v0 takes three steps: get the tuple, put the attribute in a primitive\_value wrapper, put the attribute in the tuple.

poset\_dof\_map ltuple = lposet.member\_dof\_map(lv0\_pod, true);

primitive\_value lpv(int(0));

ltuple.put\_dof(latt\_pod, lpv);

These three steps have to be done, but they don't have to be done explicitly. Implicit conversions actually make it much simpler:

lposet.member\_dof\_map(lv1\_pod, true).put\_dof(latt\_pod, int(0));

lposet.member\_dof\_map(ls0\_pod, true).put\_dof(latt\_pod, int(1));

We can also set an attribute value using either the scoped id signature:

lposet.member\_dof\_map(lv0\_pod, true).put\_dof(latt\_id, int(0));

or the name signature:

lposet.member\_dof\_map(lv0\_pod, true).put\_dof("INT", int(0));

In practice, you use whichever signature is most convenient.

#### Getting an attribute

The implicit conversions work when getting an attribute value as well:

int ldim = lposet.member\_dof\_map(lv0\_pod, false).dof(latt\_pod);

As with the mutator, the accessor works with scoped id or name:

int ldim = lposet.member\_dof\_map(lv0\_pod, false).dof(latt\_id);

int ldim = lposet.member\_dof\_map(lv0\_pod, //false).dof("INT");

### Creating join reducible members

We can create jrms and link them up at any time, whether we're in jim\_edit\_mode or not. To create a jrm, just set the xis\_jim argument to false in new\_member.

poset\_index\_type lc0\_pod = lposet.new\_member(false);

lposet.put\_member\_name(lc0\_pod, "c0", true);

Then link it up:

lposet.new\_link(ls0\_pod, lc0\_pod);

lposet.new\_link(lc0\_pod, lv0\_pod);

lposet.new\_link(lc0\_pod, lv1\_pod);

But we can't stop here. We have to make sure the cover relation is indeed a cover relation. The segment no longer covers the two vertices, so we have to remove those links using delete\_link:

void delete\_link (pod\_index\_type xgreater, pod\_index\_type xlesser)

Delete the cover link between xgreater and xlesser.

which is pretty straight forward:

lposet.delete\_link(ls0\_pod, lv0\_pod);

lposet.delete\_link(ls0\_pod, lv1\_pod);

Remember, when you're editing the graph, it's your job to get it right! In particular, it's your job to make sure there are no transitive links (links equivalent to a path) like the two we just removed. An invalid graph can produce subtle errors that are difficult to track down. We'll see shortly that in many cases, you can create and link up a jrm in a single step using the join operation. In that case, the system does all the graph editing and makes sure the graph is valid.

### Creating join equivalent members

What if we create another jrm, let's call it c1, and link it between s0 and c0:

pod\_index\_type lc1\_pod = lposet.new\_member(false);

lposet.delete\_link(ls0\_pod, lc0\_pod);

lposet.new\_link(ls0\_pod, lc1\_pod);

lposet.new\_link(lc1\_pod, lc0\_pod);

The result is shown in Figure 3, but what does it mean? Is it even legal, since c1 has only a single member in its lower cover, doesn't it have to be a jim?



Figure 3: A join equivalent member.

Well, to answer the first question, let's think about part space. A jrm is a composite part in the part space metaphor and a composite part is an assembly of basic parts. More specifically, it is the assembly of the basic parts in its down set, the set of all parts below it in the graph. Pretty clearly, the set of basic parts in the down set of our new jrm is exactly the same as the set of basic parts in the down set of c0. So our new jrm is a distinct member, but as an assembly it is identical to c0. In other words, the new jrm is a copy of c0. We say that c1 is join equivalent to c0 and call it a join equivalent member or "jem", pronounced like "gem".

Now for the second question: is it legal? The short answer is yes, but why this is true takes a little bit of explanation. The interested reader can find the full answer in Appendix B. For those not interested in the mathematical details, feel free to copy members whenever you please, as many times as you please. In fact, in section 11 we'll see some functions that make it easy to make copies.

### Deleting poset members

Deleting a member is the inverse of creating it. You have to unlink it, delete it, and relink the remaining members appropriately. We've already seen how to delete links. To delete a member, use the delete\_member function:

virtual void delete\_member (pod\_index\_type xindex)

Delete the member with index xindex. Warning: this routine does not delete links; it will leave any links to this member dangling.

So let's delete the jem we just created:

lposet.delete\_link(lc1\_pod, lc0\_pod);

lposet.delete\_link(ls0\_pod, lc1\_pod);

lposet.new\_link(ls0\_pod, lc0\_pod);

lposet.delete\_member(lc1\_pod);

### Example 10: Reading a sheaf file; manipulating poset members with the poset interface

We've covered the basics of creating, linking, accessing, and deleting poset members using the poset interface. Let's collect everything we've covered together into a single example. We'll learn to read a poset from a file as well.

#include "sheaves\_namespace.h"

#include "poset.h"

#include "poset\_dof\_map.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example10:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example10");

// Populate the namespace from the file we wrote in example9.

// Retrieves the simple\_poset example.

storage\_agent lsa("example9.hdf", sheaf\_file::READ\_ONLY);

lsa.read\_entire(lns);

// Get a reference to the poset "simple\_poset".

poset\_path lpath("simple\_poset");

poset& lposet = lns.member\_poset<poset>(lpath, true);

// Allow creation of jims.

lposet.begin\_jim\_edit\_mode(true);

// Create jims for the two vertices and the segment.

pod\_index\_type lv0\_pod = lposet.new\_member(true);

pod\_index\_type lv1\_pod = lposet.new\_member(true);

pod\_index\_type ls0\_pod = lposet.new\_member(true);

// Make the segment cover the vertices.

lposet.new\_link(ls0\_pod, lv0\_pod);

lposet.new\_link(ls0\_pod, lv1\_pod);

// Top covers the segment.

lposet.new\_link(TOP\_INDEX, ls0\_pod);

// The vertices cover bottom.

lposet.new\_link(lv0\_pod, BOTTOM\_INDEX);

lposet.new\_link(lv1\_pod, lposet.bottom().index().pod());

// We're finished creating and linking jims.

lposet.end\_jim\_edit\_mode();

// Give each jim a name..

lposet.put\_member\_name(lv0\_pod, "v0", true);

lposet.put\_member\_name(lv1\_pod, "v1", true);

lposet.put\_member\_name(ls0\_pod, "s0", true);

// Print the names to cout.

cout << lposet.member\_name(lv0\_pod) << endl;

cout << lposet.member\_name(lv1\_pod) << endl;

cout << lposet.member\_name(ls0\_pod) << endl;

// Get the row attribute id space and pod and

// scoped ids for the only attribute.

const index\_space\_handle& latt\_id\_space = lposet.schema().dof\_id\_space(false);

pod\_index\_type latt\_pod = lposet.schema().dof\_id\_space(false).begin();

scoped\_index latt\_id(lposet.schema().dof\_id\_space(false), latt\_pod);

// Get the attribute tuple for vertex 0.

poset\_dof\_map& ltuple = lposet.member\_dof\_map(lv0\_pod, true);

// Set the only attribute of v0 to its dimension, 0.

// Do the first one explicitly, without any automatic conversion.

primitive\_value lpv(int(0));

ltuple.put\_dof(latt\_pod, lpv);

// Set attributes for v1 and s0 relying on conversions.

lposet.member\_dof\_map(lv1\_pod, true).put\_dof(latt\_pod, int(0));

lposet.member\_dof\_map(ls0\_pod, true).put\_dof(latt\_pod, int(1));

// Get attributes back and write them to cout.

int lv0\_dim = lposet.member\_dof\_map(lv0\_pod, false).dof(latt\_pod);

int lv1\_dim = lposet.member\_dof\_map(lv1\_pod, false).dof(latt\_pod);

int ls0\_dim = lposet.member\_dof\_map(ls0\_pod, false).dof(latt\_pod);

cout << "v0 dim= " << lv0\_dim;

cout << " v1 dim= " << lv1\_dim;

cout << " s0 dim= " << ls0\_dim;

cout << endl;

// Create a jrm named c0.

pod\_index\_type lc0\_pod = lposet.new\_member(false);

lposet.put\_member\_name(lc0\_pod, "c0", true);

// Link it up:

lposet.new\_link(ls0\_pod, lc0\_pod);

lposet.new\_link(lc0\_pod, lv0\_pod);

lposet.new\_link(lc0\_pod, lv1\_pod);

// Delete the now obsolete links from s0 to the vertices.

lposet.delete\_link(ls0\_pod, lv0\_pod);

lposet.delete\_link(ls0\_pod, lv1\_pod);

// Create a jem; a copy of c0, call c1.

pod\_index\_type lc1\_pod = lposet.new\_member(false);

lposet.put\_member\_name(lc1\_pod, "c1", true);

lposet.delete\_link(ls0\_pod, lc0\_pod);

lposet.new\_link(ls0\_pod, lc1\_pod);

lposet.new\_link(lc1\_pod, lc0\_pod);

// Output the finished poset to cout:

cout << lposet << endl;

// Delete c1.

lposet.delete\_link(lc1\_pod, lc0\_pod);

lposet.delete\_link(ls0\_pod, lc1\_pod);

lposet.new\_link(ls0\_pod, lc0\_pod);

lposet.delete\_member(lc1\_pod);

// Exit:

return 0;

}

Make sure to run example9 before example10 and in the same directory so that example10 can find the file example9.hdf. The output from example10 is in the file example10.cout

## Poset member handles

The poset member class hierarchy provides an alternate interface for manipulating poset members. Abstract\_poset\_member is the abstract base class for the hierarchy, with immediate descendants partial\_poset\_member and total\_poset\_member. The partial/total adjective refers to whether the interface supports restriction of a member to only part of its schema; partial\_poset\_member does and total\_poset\_member doesn't. Partial\_poset\_member is the base class for the various types of sections in the section\_spaces cluster of the fiber\_bundles component, for which restriction is an important operation. Total\_poset\_member is the type of member for ordinary posets and is the base class for the various algebraic type in the fiber\_spaces cluster in fiber\_bundles, for which restriction doesn't really make much sense.

The most prominent feature of abstract\_poset\_member and its descendants it that they are handles. We've already seen index space handles. Poset member handles are similar in concept but are used somewhat differently. First, poset member handle are not stored in pools managed by the system; the client directly creates and destroys them. The default constructor creates an unattached handle (all the following functions are direct or inherited members of total\_poset\_member):

total\_poset\_member()

Default constructor; creates a new, unattached total\_poset\_member handle.

Once created, you can "attach" a handle to a state:

void attach\_to\_state(const poset\_state\_handle \* xhost, pod\_index\_type xindex)

Attach this handle to the state with index xindex in the current version of host xhost.

The attach\_to\_state function is available in several signatures for specifying the member to attach to by pod id, scoped id, name and some other variations, see the reference documentation.

You can combine the construction and attachment into a single step, also available in several signatures, for instance:

total\_poset\_member(const poset\_state\_handle \* xhost, pod\_index\_type xindex)

Creates a new total\_poset\_member handle attached to the member state with index xindex in xhost.

You detach a handle, so it is unattached again.

void poset\_component::detach\_from\_state()

Detach this handle from its state, if any.

You can create a handle and a new jim state in a single step:

total\_poset\_member(poset\_state\_handle\* xhost, poset\_dof\_map\* xdof\_map=0, bool

xcopy\_dof\_map=false, bool xauto\_access=true)

Creates a new jim (join-irreducible member) attached to a new member state in xhost.

You can use an existing handle to create a new state, as usual with several signatures, for instance:

void new\_jim\_state(poset\_state\_handle\* xhost, poset\_dof\_map\* xdof\_map = 0,

bool xcopy\_dof\_map = false, bool xauto\_access = true )

Creates a new jim (join-irreducible member) state in xhost and attaches this to it.

The poset member classes provides most of the operations we've already seen in the poset interface. For instance:

void create\_cover\_link(abstract\_poset\_member\* xlesser)

Insert a link from this to lesser; make lesser <= this.

and

void delete\_cover\_link(abstract\_poset\_member\* lesser)

Delete the link from this to lesser; make lesser incomparable to this.

In addition, the poset member classes provide several operations not available (yet) in the poset interface. In particular, it provides the lattice algebra operations, join and meet:

total\_poset\_member\* total\_poset\_member::l\_join(abstract\_poset\_member\* other,

bool xnew\_jem = true)

Lattice join of this with other, auto-allocated version. The lattice join is the least upper bound in the lattice generated by the jims in the poset.

total\_poset\_member\* sheaf::total\_poset\_member::l\_meet(abstract\_poset\_member\* other,

bool xnew\_jem = true)

Lattice meet of this with other, auto-allocated version. The lattice meet is the greatest lower bound in the lattice generated by the jims in the poset.

### Example 11: Manipulating poset member with the poset\_member interface.

We provide examples of the poset member handle interface by redoing Example 10 using the poset member interface instead of the poset interface.

#include "sheaves\_namespace.h"

#include "poset.h"

#include "poset\_dof\_map.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

#include "total\_poset\_member.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example11:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example11");

// Populate the namespace from the file we wrote in example9.

// Retrieves the simple\_poset example.

storage\_agent lsa("example9.hdf", sheaf\_file::READ\_ONLY);

lsa.read\_entire(lns);

// Get a reference to the poset "simple\_poset".

poset\_path lpath("simple\_poset");

poset& lposet = lns.member\_poset<poset>(lpath, true);

// Create an unattached handle.

total\_poset\_member lmbr;

cout << "lmbr is\_attached() = " << boolalpha << lmbr.is\_attached();

cout << endl;

// Attach it to the top member of our poset.

lmbr.attach\_to\_state(&lposet, TOP\_INDEX);

cout << "lmbr attached to " << lmbr.name() << endl;

// Reattach it to the bottom member.

lmbr.attach\_to\_state(&lposet, BOTTOM\_INDEX);

cout << "lmbr attached to " << lmbr.name() << endl;

// Unattach it.

lmbr.detach\_from\_state();

cout << "lmbr is\_attached() = " << lmbr.is\_attached() << endl;

// Allow creation of jims.

lposet.begin\_jim\_edit\_mode(true);

// Create jims for the two vertices and the segment.

total\_poset\_member lv0(&lposet);

total\_poset\_member lv1(&lposet);

total\_poset\_member ls0(&lposet);

// Make the segment cover the vertices.

ls0.create\_cover\_link(&lv0);

ls0.create\_cover\_link(&lv1);

// Top covers the segment.

lposet.top().create\_cover\_link(&ls0);

// The vertices cover bottom.

lv0.create\_cover\_link(&lposet.bottom());

lv1.create\_cover\_link(&lposet.bottom());

// We're finished creating and linking jims.

lposet.end\_jim\_edit\_mode();

// Give each jim a name..

lv0.put\_name("v0", true, true);

lv1.put\_name("v1", true, true);

ls0.put\_name("s0", true, true);

// Print the names to cout.

cout << lv0.name() << endl;

cout << lv1.name() << endl;

cout << ls0.name() << endl;

// Get the row attribute id space and pod and scoped ids

// for the only attribute.

const index\_space\_handle& latt\_id\_space = lposet.schema().dof\_id\_space(false);

pod\_index\_type latt\_pod = lposet.schema().dof\_id\_space(false).begin();

scoped\_index latt\_id(lposet.schema().dof\_id\_space(false), latt\_pod);

// Set attributes for v0, v1, and s0 relying on conversions.

lv0.dof\_map(true).put\_dof(latt\_pod, int(0));

lv1.dof\_map(true).put\_dof(latt\_pod, int(0));

ls0.dof\_map(true).put\_dof(latt\_pod, int(1));

// Get attributes back and write them to cout.

int lv0\_dim = lv0.dof\_map(false).dof(latt\_pod);

int lv1\_dim = lv1.dof\_map(false).dof(latt\_pod);

int ls0\_dim = ls0.dof\_map(false).dof(latt\_pod);

cout << "v0 dim= " << lv0\_dim;

cout << " v1 dim= " << lv1\_dim;

cout << " s0 dim= " << ls0\_dim;

cout << endl;

// Create a jrm named c0.

// C0 is the join of v0 and v1, so we can create it

// and link it up in a single step using the join operator.

total\_poset\_member\* lc0 = lv0.l\_join(&lv1, false);

lc0->put\_name("c0", true, true);

// Create a jem; a copy of c0, Call it c1.

total\_poset\_member lc1(\*lc0, true);

lc1.put\_name("c1", true, true);

// Output the finished poset to cout:

cout << lposet << endl;

// Delete c1.

lc1.delete\_state(true);

cout << "c1 is\_attached() = " << boolalpha << lc1.is\_attached();

cout << endl;

// Exit:

return 0;

}

## Subposets

A subposet is, as the name suggests, a subset of a poset. We've already seen that subsets can be represented by id spaces, so why subposets? The answer is mostly historical, the id space concept emerged as a generalization of subposets and id maps. The id maps no longer exist as independent entities, they have been subsumed by the id spaces, but subposet lives on, at least for a while.

A subposet is essentially a bit vector with a bit for each member of the poset. The bit corresponding to a member is set (true) if the member of the poset is a member of the subposet. The bit vector representation is efficient in both time and memory for subsets that contain more than a few percent of the total poset, which is one reason the subposet class is still used.

The other reasons subposet is still around is that it is used for three critical roles in the system. The first critical role is the "whole" subposet. The whole subposet defines which ids are actually members of the poset; when a member is deleted the corresponding id continues to exist, but the member is no longer in the whole subposet. The functionality of whole subposet has been largely replaced by the member\_hub\_id\_space.

The second critical role for subposet is the jims subposet. A member is a jim if and only if it is in the jims subposet. Finally, as we will learn shortly, subposets are used as filters in depth first searches of the graph.

Whenever possible, clients should use id spaces instead of subposets.

The following example shows the basic features of subposets.

### Example 12: Subposets

#include "sheaves\_namespace.h"

#include "poset.h"

#include "poset\_dof\_map.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

#include "subposet.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example12:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example12");

// Populate the namespace from the file we wrote in example9.

// Retrieves the simple\_poset example.

storage\_agent lsa\_read("example10.hdf", sheaf\_file::READ\_ONLY);

lsa\_read.read\_entire(lns);

// Get a reference to the poset "simple\_poset".

poset\_path lpath("simple\_poset");

poset& lposet = lns.member\_poset<poset>(lpath, true);

// Create a subposet called "jrms".

subposet ljrms(&lposet);

ljrms.put\_name("jrms", true, false);

// Test to see if it is empty (constructor ensures that it is).

cout << "subposet " << ljrms.name();

cout << " is empty()= " << boolalpha << ljrms.is\_empty();

cout << endl;

// Put top, c1, c0, and bottom into the subposet.

// Use all the different signatures for insert\_member.

scoped\_index ltop\_id(lposet.top().index());

ljrms.insert\_member(ltop\_id);

pod\_index\_type lc1\_pod = lposet.member\_id("c1", false);

ljrms.insert\_member(lc1\_pod);

pod\_index\_type lc0\_pod = lposet.member\_id("c0", false);

ljrms.insert\_member(lc0\_pod);

abstract\_poset\_member\* lbot = &lposet.bottom();

ljrms.insert\_member(lbot);

// Check if it contains the member we just put in.

cout << "contains top: " << ljrms.contains\_member(ltop\_id) << endl;

cout << "contains c1: " << ljrms.contains\_member(lc1\_pod) << endl;

cout << "contains c0: " << ljrms.contains\_member(lc0\_pod) << endl;

cout << "contains bottom: " << ljrms.contains\_member(lbot) << endl;

// Get an iterator for the members and print out their names.

cout << "Subposet jrms contains:";

index\_iterator litr = ljrms.indexed\_member\_iterator();

while(!litr.is\_done())

{

// Print out the member name.

cout << " " << lposet.member\_name(litr.index(), false);

litr.next();

}

cout << endl;

// Remove top and bottom.

ljrms.remove\_member(lposet.top().index());

ljrms.remove\_member(&lposet.bottom());

// Print out the member names again.

cout << "Subposet jrms contains:";

litr.reset();

while(!litr.is\_done())

{

// Print out the member name.

cout << " " << lposet.member\_name(litr.index(), false);

litr.next();

}

cout << endl;

// Exit:

return 0;

}

## Traversing the graph

Once you've constructed a poset or retrieved one from storage, you typically want to move around in it, explore it. The requires traversing (searching) the covering relation graph and the SheafSystem provides iterators for that purpose. There are two kinds cover iterators and depth-first iterators.

### Cover id spaces and iterators

Local searches, in the immediate neighborhood of a given poset member are accomplished with cover iterators. Remember that every member except bottom has a set of outgoing links to the lower cover of the member and every member except top has a set of incoming links to its upper cover. The lower and upper cover of each member are represented as id spaces:

index\_space\_handle& poset\_state\_handle::

get\_cover\_id\_space(bool xlower, pod\_index\_type xmbr\_hub\_id) const

Allocates a handle for the lower (xlower true) or upper (xlower false) cover id space of the member with hub id xmbr\_hub\_id from the pool of id spaces.

As always, if you get\_ it you have to release\_ it when you're finished with it.

void release\_cover\_id\_space (index\_space\_handle &xcover\_id\_space) const

Returns xcover\_id\_space to the pool of id spaces.

We can get the id space from the poset and get/release an iterator from the id space or get/release the iterator directly from the poset:

index\_space\_iterator& poset\_state\_handle::

get\_cover\_id\_space\_iterator(bool xlower, pod\_index\_type xmbr\_hub\_id) const

Allocates an iterator for the lower (xlower true) or upper (xlower false) cover id space of the member with hub id xmbr\_hub\_id from the pool of id space iterators.

void poset\_state\_handle::

release\_cover\_id\_space\_iterator (index\_space\_iterator &xcover\_itr) const

Returns xcover\_itr to the pool of id space iterators.

The cover\_id\_space and cover\_id\_space\_iterator functions are also available from poset member handles.

#### Example 13: Cover id spaces and iterators

#include "index\_space\_handle.h"

#include "index\_space\_iterator.h"

#include "poset.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

#include "total\_poset\_member.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example12:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example12");

// Populate the namespace from the file we wrote in example10.

// Retrieves the simple\_poset example.

storage\_agent lsa\_read("example10.hdf", sheaf\_file::READ\_ONLY);

lsa\_read.read\_entire(lns);

// Get a reference to the poset "simple\_poset".

poset\_path lpath("simple\_poset");

poset& lposet = lns.member\_poset<poset>(lpath, true);

// Get the hub id for member "c0".

pod\_index\_type lc0\_pod = lposet.member\_id("c0", false);

// Get an iterator for the lower cover to member "c0".

index\_space\_handle& lc0\_lc1 = lposet.get\_cover\_id\_space(true, lc0\_pod);

index\_space\_iterator& lc0\_lc\_itr1 = lc0\_lc1.get\_iterator();

// Iterate over the members of the lower cover.

cout << "Lower cover of c0 is:";

while(!lc0\_lc\_itr1.is\_done())

{

cout << " " << lposet.member\_name(lc0\_lc\_itr1.hub\_pod());

lc0\_lc\_itr1.next();

}

cout << endl;

// We're finished with the iterator, release it.

lc0\_lc1.release\_iterator(lc0\_lc\_itr1);

// Get an iterator for the upper cover of c0.

// The enums LOWER and UPPER are defined in the sheaf namespace

// true and false, respectively.

index\_space\_iterator& lc0\_uc\_itr1 = lposet.get\_cover\_id\_space\_iterator(UPPER, lc0\_pod);

// Iterate over the members of the upper cover.

cout << "Upper cover of c0 is:";

while(!lc0\_uc\_itr1.is\_done())

{

cout << " " << lposet.member\_name(lc0\_uc\_itr1.hub\_pod());

lc0\_uc\_itr1.next();

}

cout << endl;

// We're finished with the iterator, release it.

lposet.release\_cover\_id\_space\_iterator(lc0\_uc\_itr1);

// Repeat all the above using the poset member handle interface.

// Get a handle for member c0.

total\_poset\_member lc0\_mbr(&lposet, "c0");

// Get an iterator for the lower cover to member "c0".

index\_space\_handle& lc0\_lc2 = lc0\_mbr.get\_cover\_id\_space(true);

index\_space\_iterator& lc0\_lc\_itr2 = lc0\_lc2.get\_iterator();

// Iterate over the members of the lower cover.

cout << "Lower cover of c0 is:";

while(!lc0\_lc\_itr2.is\_done())

{

cout << " " << lposet.member\_name(lc0\_lc\_itr2.hub\_pod());

lc0\_lc\_itr2.next();

}

cout << endl;

// We're finished with the iterator, release it.

lc0\_lc2.release\_iterator(lc0\_lc\_itr2);

// Get an iterator for the upper cover of c0.

// The enums LOWER and UPPER are defined in the sheaf namespace

// true and false, respectively.

index\_space\_iterator& lc0\_uc\_itr2 = lc0\_mbr.get\_cover\_id\_space\_iterator(UPPER);

// Iterate over the members of the upper cover.

total\_poset\_member luc\_mbr;

cout << "Upper cover of c0 is:";

while(!lc0\_uc\_itr2.is\_done())

{

luc\_mbr.attach\_to\_state(&lposet, lc0\_uc\_itr2.hub\_pod());

cout << " " << luc\_mbr.name();

lc0\_uc\_itr2.next();

}

cout << endl;

// We're finished with the iterator, release it.

luc\_mbr.release\_cover\_id\_space\_iterator(lc0\_uc\_itr2);

// Exit:

return 0;

}

### Depth first traversal

Global searches, over the larger portions of a graph than the immediate neighborhood of a single member are accomplished using the depth\_first\_itr family of iterators. As implied by the name, these iterators execute a depth first search of the graph. Readers unfamiliar with depth first search should consult a basic computer science text such as Aho and Ulman "Foundations of Computer Science".

A depth first iterator starts from a given member called the anchor for the iteration and transitively follows all the links in a given direction, either down or up. A depth first search always visits all the children of a node before it visits any of the siblings of the node. There are three locations relative to a given member p where some action may be performed in a depth first search, as shown in Figure 4. The previsit action occurs immediately before visiting the first child, so it is the first opportunity for acting on p. The postvisit action occurs after all the children have been visited, so it is the last action opportunity for p with p as the current member of the iteration. The link action for a given child occurs "on the link", immediately after visiting the child. The link action for a child, with the parent as current member, occurs immediately after the postvisit action for the child, with the child as current member. So a link action can still affect a node after the postvisit action on that node has occurred.



Figure 4: Actions in depth first search.

The members of the depth\_first\_itr hierarchy differentiate on which of the action opportunities are exposed to the client. Class preorder\_itr returns control to the user only in the previsit position, immediately before iterating over the links. Class postorder\_itr returns control to the client only in the postvisit position, after visiting all the children. Class biorder\_itr returns control in both the previsit and postvisit positions. Class linkorder\_itr exposes the link action opportunities. Class triorder\_itr exposes all the action opportunities.

The postorder\_itr is the most frequently used iterator. Postorder has the extremely useful property that by the time a given node is visited, all the nodes less than it in the order relation have been visited.

The preorder\_itr is the next most frequently used. It is useful because the depth first search can be truncated from the preorder position. Instead of visiting all the children, transitively, iteration can forced to jump over the children to the next sibling. This is useful for finding maximal and minimal members of some subset, as we'll see in the example.

Biorder\_itr, linkorder\_itr, and triorder\_itr are useful for advanced algorithms, especially iterations that change the graph.

All five of these classes are actually class templates, for instance:

template<typename T> class sheaf::postorder\_itr< T >

Specialization of the filtered depth-first iterator which exposes the POSTVISIT\_ACTION to the client.

The parameter specifies the type of data structure used to track which nodes in the graph have already been visited. Three specializations are provided: zn\_to\_bool (a bit vector type), set, and hash set. The specializations are provided because they have different performance characteristics. Let N be the number of nodes in the graph and M be the number visited in a given traversal. Zn\_to\_bool uses O(1) time to check if a node has been visited already, but uses one bit per node and hence uses O(N) memory, independent of how much of the graph is begin searched. Zn\_to\_bool is most efficient for traversal of the entire graph, but the O(N) memory means it takes O(N) time to initialize the marker structure, a serious problem for repeated small searches. Set uses O(log(M)) time to check if a node has been visited and O(M) memory, so it is better for small searches. Hash\_set uses O(1) time and O(M) memory, so it is very good for small searches, but uses more memory than zn\_to\_bool when applied to the whole graph. Typedefs are provided for the three specializations, for instance:

typedef postorder\_itr<zn\_to\_bool> sheaf::zn\_to\_bool\_postorder\_itr

Postorder\_itr<T> using zn\_to\_bool for \_has\_visited markers.

typedef postorder\_itr< set<pod\_index\_type> > sheaf::set\_postorder\_itr

Postorder\_itr<T> using set for \_has\_visited markers.

typedef postorder\_itr< hash\_set<pod\_index\_type> > sheaf::hash\_set\_postorder\_itr

Postorder\_itr<T> using hash\_set for \_has\_visited markers.

When we create an iterator, we specify the anchor, a filter, the search direction (down or up), and strictness (whether the anchor itself is visited):

postorder\_itr(const abstract\_poset\_member & xanchor, pod\_index\_type xfilter\_index,

bool xdown, bool xstrict)

Creates an iterator anchored at xanchor, filtered by xfilter\_index. If xdown, iterate in the down direction, otherwise iterate up. If xstrict, iterate over strict up/down set only.

The filter is a subposet; the iterator only returns control to the client if the node is a member of the subposet.

#### Example 14: Depth first iterators

#include "poset.h"

#include "postorder\_itr.h"

#include "preorder\_itr.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

#include "total\_poset\_member.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example14:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example14");

// Populate the namespace from the file we wrote in example10.

// Retrieves the simple\_poset example.

storage\_agent lsa\_read("example10.hdf", sheaf\_file::READ\_ONLY);

lsa\_read.read\_entire(lns);

// Get a reference to the poset "simple\_poset".

poset\_path lpath("simple\_poset");

poset& lposet = lns.member\_poset<poset>(lpath, true);

// Create a postorder iterator for the down set of top.

// Filter with the jims subposet.

// DOWN and UP are enums for true and false, respectively.

// Include the anchor itself in the search.

zn\_to\_bool\_postorder\_itr

lpost\_itr(lposet.top(), JIMS\_INDEX, DOWN, NOT\_STRICT);

// Find all the jims.

while(!lpost\_itr.is\_done())

{

cout << lposet.member\_name(lpost\_itr.index(), false);

cout << " is a jim";

cout << endl;

lpost\_itr.next();

}

// Find just the minimal jims (atoms).

// Iterate up from the bottom and truncate when we find something.

zn\_to\_bool\_preorder\_itr

lpre\_itr(lposet.bottom(), JIMS\_INDEX, UP, NOT\_STRICT);

while(!lpre\_itr.is\_done())

{

cout << lposet.member\_name(lpre\_itr.index(), false);

cout << " is an atom";

cout << endl;

lpre\_itr.truncate();

}

// Exit:

return 0;

}

## Schema posets

We took a short cut in section 9.2 "Creating posets" to avoid having to create a schema poset before we knew how to create an ordinary poset. Now that we've seen the basics of creating posets and poset members, we can return to the task of creating a schema poset.

Actually, there is not much special about a schema poset. It is just an ordinary poset, except the schema of a schema poset has to be the "schema schema", the top member of the primitives\_schema poset, and it has to have a couple of specific subposets created in it. But the programmer doesn't have to worry about these details because the namespace\_poset has a factory method for making schema posets:

poset & new\_schema\_poset (const string &xname, bool xauto\_access)

Creates a new schema poset with name xname.

So let's create the cell schema that was used for the line segment example in the Part Space tutorial. The cell schema is shown in Figure 5, which reproduces Figure 14 from the Part Space tutorial.

Using the factory method, we can create the cell schema poset in a single line:

poset& lcell\_schema = lns.new\_schema\_poset("cell\_schema", true);

Once we've got the schema poset, it's time to create some members. The class schema\_poset\_member is a specialized poset member handle for creating and manipulating schema members. Typically the most convenient way to construct a schema member is with:



Figure 5: The cell schema table copied from the Part Space tutorial, Figure 14.

schema\_poset\_member(const namespace\_poset &xns, const string &xname,

const poset\_path &xparent\_path,

const wsv\_block<schema\_descriptor> &xdof\_specs,

bool xschematize,

bool xauto\_access=true)

Creates a new jim which conforms\_to the schema with path xparent\_path and has additional dofs with names, types, and roles specified by xdof\_specs. Xschematize is obsolete and ignored.

This constructor supports single inheritance. The xparent\_path argument specifies the path to the base class for the type being defined. The xdof\_space argument specifies any additional data members. Class schema\_descriptor is essentially a struct with members for the name, type, and is\_tbl properties of an attribute. The "wsv" in wsv\_block stands for "whitespace separated value" and the wsv\_block template is essentially an array with features for easily entering elements as strings. Both define appropriate constructors and implicit conversions to make it easy to define the schema for individual attributes. The easiest way to explain how they work is with an example. Recall the cell class hierarchy from the Part Space tutorial:

class cell: public spatial\_structure

{

string cell\_type; // The type of spatial cell

}

class spatial\_structure

{

int d; // The spatial dimension of the structure.

}

Class cell inherits class spatial\_structure, so first we create the spatial\_structure schema. It doesn't inherit anything, so we specify bottom as the parent. Poset\_path has a constructor that takes a string literal, so we can specify the path using a string literal and rely on implicit conversion to poset\_path. Spatial\_structure has one data member, name "d", type int, not a table attribute. The wsv\_block<schema\_descriptor> combination lets you just string the attribute specifications together, separated by white space. The set of possible primitives types is specified in the enum primitive\_type in file primitive\_types.h in the include directory of the SheafSystem installation. The enumerator for type int is "INT". So the spec for the only data member of spatial structure is "d INT false". For more attributes, just string them together, separated by whitespace. So the call to create spatial\_structure is:

schema\_poset\_member lspatial(lns, "spatial\_structure", "cell\_schema\_poset/bottom", "d INT false", false, true);

Cell inherits spatial\_structure and adds one data member, name cell\_type, type string, not a table attribute. The primitive type corresponding to a C++ string object is a C string, with enumerator C\_STRING, so we create the cell schema with:

schema\_poset\_member lcell(lns, "cell", lspatial.path(),

"cell\_type C\_STRING false", false, true);

So now we've created the schema for the spatial\_structure and cell types, with cell inheriting spatial\_structure. Let's verify that cell inherits spatial\_structure. We do that with:

bool sheaf::schema\_poset\_member::

conforms\_to(const schema\_poset\_member & xother) const

True if the dofs defined by this agree in type and in order with the dofs defined by xother. (This schema may contain additional dofs as well.)

So we can test whether lcell inherits lspatial with:

cout << "cell conforms to spatial\_structure= ";

cout << boolalpha << lcell.conforms\_to(lspatial);

cout << endl;

Each schema member defines an id space for its row attributes and an id space for its table attributes. We can use ids from the attribute id space to access the properties of the attributes using, for instance:

sheaf::size\_type sheaf::schema\_poset\_member::

size(pod\_index\_type xdof\_id, bool xis\_table\_dof) const

The number of bytes in the table dof (xis\_table\_dof true) or row dof referred to by xdof\_id in the schema defined by this.

There are similar accessors for name, type, alignment, and offset.

We demonstrate the use of all of these features and more in Example 15.

### Example 15: Schema poset

#include "index\_space\_iterator.h"

#include "schema\_poset\_member.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

#include "wsv\_block.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example15:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example15");

// Create the cell schema poset.

poset& lposet = lns.new\_schema\_poset("cell\_schema\_poset", true);

// Create the schema for spatial\_structure.

// It doesn't inherit anything, so specify bottom as the parent.

// It has one data member, name "d", type "INT", not a table

// attribute

schema\_poset\_member lspatial(lns, "spatial\_structure",

"cell\_schema\_poset/bottom", "d INT false", false, true);

// Cell inherits spatial\_structure and adds one data member,

// name cell\_type, type C\_STRING, not a table attribute.

schema\_poset\_member lcell(lns, "cell", lspatial.path(),

"cell\_type C\_STRING false", false, true);

// Verify that cell inherits spatial\_structure.

cout << endl;

cout << "cell conforms to spatial\_structure= ";

cout << boolalpha << lcell.conforms\_to(lspatial);

cout << endl;

cout << endl;

// Get an iterator for the row attribute id space of the cell schema.

index\_space\_iterator& litr =

lcell.dof\_id\_space(false).get\_iterator();

while(!litr.is\_done())

{

// Print attribute info.

pod\_index\_type latt\_pod = litr.pod();

cout << "name: " << lcell.name(latt\_pod, false);

cout << " size: " << lcell.size(latt\_pod, false);

cout << " alignment: " << lcell.alignment(latt\_pod, false);

cout << " type: " << lcell.type(latt\_pod, false);

cout << " offset: " << lcell.offset(latt\_pod, false);

cout << endl;

litr.next();

}

lcell.dof\_id\_space(false).release\_iterator(litr);

// Print out the schema.

cout << lposet << endl;

// Test the schema by creating a poset using it.

poset& ltest =

lns.new\_member\_poset<poset>("test", lcell.path(), "", true);

ltest.begin\_jim\_edit\_mode(true);

pod\_index\_type lmbr0 = ltest.new\_member(true);

pod\_index\_type lmbr1 = ltest.new\_member(true);

pod\_index\_type lmbr2 = ltest.new\_member(true);

ltest.end\_jim\_edit\_mode(true, true);

ltest.put\_member\_name(lmbr0, "v0", true, false);

ltest.member\_dof\_map(lmbr0, true).put\_dof("d", int(0));

ltest.member\_dof\_map(lmbr0, true).put\_dof("cell\_type", "vertex");

ltest.put\_member\_name(lmbr1, "v1", true, false);

ltest.member\_dof\_map(lmbr1, true).put\_dof("d", int(0));

ltest.member\_dof\_map(lmbr1, true).put\_dof("cell\_type", "vertex");

ltest.put\_member\_name(lmbr2, "s0", true, false);

ltest.member\_dof\_map(lmbr2, true).put\_dof("d", int(1));

ltest.member\_dof\_map(lmbr2, true).put\_dof("cell\_type", "segment");

cout << ltest << endl;

// Exit:

return 0;

}

skip example ids: Example 16Example 17Example 18Example 19Example 20

# Part II: The fiber bundle component

The sheaf component provides the fundamental mechanism for defining arbitray persistent data types. The fiber bundle compoent uses this capability to define the types of the fiber bundle data model: base spaces, fiber spaces, and section spaces. We'll describe the basic functionality provided by the SheafSystem for each of these roles.

## Fiber\_bundles\_namespace

The fiber\_bundles\_namespace class provides the sheaf schema for the specialized types defined by the fiber bundle component. As with the sheaf component, the first step in using the fiber buundle component is to create a namespace object, but for the fiber bundle component, it must be a fiber\_bundles\_namespace. You create a fiber\_bundles\_namespace object in the same way you create a sheaves\_namespace, just give the constructor a name:

fiber\_bundles\_namespace(const string &xname)

Creates a fiber bundles namespace with name xname.

For example:

fiber\_bundles\_namespace lns("example16");

For the remainder of Part II, any namespace we mention is a fiber\_bundles\_namespace.

## Base spaces

As described in the Part Spaces For Scientific Computing tutorial and the Sheaf System Analsysis and Design Tutorial, the base space role in a property association is typically represented by a mesh. A mesh is a decomposition of some domain into a collection of smaller parts for the purpose of representing the dependence of one or more properties on position within the domain. In the sheaf system, a mesh is represented as aposet, with a jim for each of the basic parts in the decomposition. As with all tables in the SheafSystem, a base space table needs a schema.

### Base space schema

Our discussion of the sheaf component delayed discussion of schema posets to the end of the discussion because we had to know how about posets in general before discussing schema posets. But you need a schema poset before you can create an object poset, so from here on we discuss the schema up front.

The fiber\_bundle\_bundles namespace defines a single universal schema for any base spaces. The base space schema poset is accessible from the namespace:

poset& fiber\_bundles\_namespace::base\_space\_schema\_poset()

The poset defining the schemas for base\_space\_member and descendants.

However, the programmer rarely needs to even be aware of this poset, much less use it, because the system takes care of it automatically.

### Base space posets

Meshes are represented using the class base\_space\_poset. A base\_space\_poset inherits poset, so it is\_a poset, but the class has several member functions useful for manipulating meshes. The namespace provides a factory method for creating base space posets:

template <typename B>

base\_space\_poset& new\_base\_space(const poset\_path& xbase\_space\_path,

const arg\_list& xargs = "",

const poset\_path& xschema\_path = "",

int xmax\_db = -1,

bool xauto\_access = true);

Find or create a new base space for meshes (blocks) of type B.

The function is templated on the kind of members the mesh will contain. The type base\_space\_member\_prototype can be used to create a poset that will accept any other type of member, we' discuss those shortly.

The only argument that must always be provided to new\_base\_space is the name, in the form of a poset path. If the other arguments are not provided, the new\_base\_space function will apply defaults appropriate for the type of member the poset is being created for.

An exception to depending on default values is that if the maximum cell dimension B::DB specified by the member type B is -1, then max\_db, must be provided either through xargs or directly with the xmax\_db argument. The value for DB is specified in the reference documentation for each member type.

The reason for the additional arguments is that the system is designed to be extensible. You can program derived classes that inherit class base\_space\_poset. and such specializations may require arguments for the constructor, hence the xargs argument. Similarly, you can define specialized schema, even for class base\_space\_poset, hence the xschema\_path argument. But these are advanced topics we won't cover in this introduction.

An example will show how this works. Let's return to the line segment example we used in the discussion of the sheaf component and re-do it as a base space poset. To create a base space with schema general enough for any type of member, specify base\_space\_member\_prototype as the member type. Consulting the reference documentation for base\_space\_member\_prototype, we find DB = -1. So we have to provide max\_db; we'll use xargs argument. Like every poset, base\_space\_poset provides the make\_args function:

static sheaf::arg\_list fiber\_bundle::base\_space\_poset::make\_args(int xmax\_db)

Makes a constructor arg\_list for an instance with maximum intrinsic dimension xmax\_db. Intended for use with fiber\_bundles\_namespace::new\_base\_space.

For our line segment example the maximum dimension is 1, so:

arg\_list largs = base\_space\_poset::make\_args(1);

base\_space\_poset& lbsp=

lns.new\_base\_space<base\_space\_member\_prototype>("mesh", largs);

### Example 21: Creating a base\_space\_poset.

#include "base\_space\_member\_prototype.h"

#include "base\_space\_poset.h"

#include "fiber\_bundles\_namespace.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

using namespace sheaf;

using namespace fiber\_bundle;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example21:" << endl;

// Create a namespace.

fiber\_bundles\_namespace lns("Example21");

// We use a schema with a single integer attribute.

poset\_path lschema\_path("primitives", "INT");

// The contructor for base\_space\_poset needs

// max\_db = 1 for line\_segment mesh.

arg\_list largs = base\_space\_poset::make\_args(1);

// Create the poset.

base\_space\_poset& lposet =

lns.new\_base\_space<base\_space\_member\_prototype>("mesh", largs);

// Print the poset to cout.

cout << lposet << endl;

// Write the namespace to a sheaf file.

storage\_agent lsa("example21.hdf");

lsa.write\_entire(lns);

// Exit:

return 0;

}

### Base space members

Base space members can be created in two ways. They can be created individually, like we did previously in our simple\_poset example. However, this is tedious for anything larger than the tiniest meshes and, worse, the result makes inefficient use of memory. So the system provides "block" members. A block is a collection of cells, all of the same type. We'll describe creating individual members in this section. Blocks can only be created with the handle interface, so we'll discuss them in section 5.2.5.

A base\_space\_poset is\_a poset so we can create individual members just like we did previously for the "simple\_poset" example. The only difference is that base\_space\_psoet provides special support for creating and initializing the attribute tuples for standard cell types.

const sheaf::scoped\_index&

fiber\_bundle::base\_space\_poset::

new\_row\_dof\_map(const string & xprototype\_name)

Creates a new dof map with dofs equal to those of the prototype with name xprototype\_name.

The names of the prototypes for individual members defined by the base\_space\_member\_prototypes poset in the fiber\_bundles\_namespace are shown in Table 1.

Table 1: Standard prototypes for individual cells

|  |  |  |  |
| --- | --- | --- | --- |
| 0-cells | 1-cells | 2-cells | 3-cells |
| point | segment | triangle | tetra |
|  |  | quad | hex |
|  |  | general\_polygon | general\_polyhedron |

Any of these can be used with new\_row\_dof\_map to create a pre-initialized attribute tuple for the named type of cell. As an example, we recreate the line segment mesh using base\_space\_poset.

#### Example 22: Base space members using the base\_space\_poset interface.

#include "base\_space\_poset.h"

#include "fiber\_bundles\_namespace.h"

#include "poset\_dof\_map.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

using namespace sheaf;

using namespace fiber\_bundle;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example22:" << endl;

// Create a namespace.

fiber\_bundles\_namespace lns("Example22");

// Populate the namespace from the file we wrote in example21.

// Retrieves the simple\_poset example.

storage\_agent lsa\_read("example21.hdf", sheaf\_file::READ\_ONLY);

lsa\_read.read\_entire(lns);

// Get a reference to the poset "mesh".

poset\_path lpath("mesh");

base\_space\_poset& lposet =

lns.member\_poset<base\_space\_poset>(lpath, true);

// Create attribute tuples for member prototypes

//"vertex" and "segment". These will be autmatically

// initialized with all the proper attribute values.

scoped\_index lv\_tuple\_id = lposet.new\_row\_dof\_map("point");

scoped\_index ls\_tuple\_id = lposet.new\_row\_dof\_map("segment");

// Allow creation of jims.

lposet.begin\_jim\_edit\_mode(true);

// Create the first vertex and assign its attributes at the same time

// by providing the attribute tuple to use.

pod\_index\_type lv0\_pod =

lposet.new\_member(true, lv\_tuple\_id.hub\_pod());

lposet.put\_member\_name(lv0\_pod, "v0", true);

// Create the second vertex with the same attribute tuple.

// v0 and v1 share the tuple; essential memory savings

// for large meshes.

pod\_index\_type lv1\_pod =

lposet.new\_member(true, lv\_tuple\_id.hub\_pod());

lposet.put\_member\_name(lv1\_pod, "v1", true);

// Create the segment with the segment tuple.

pod\_index\_type ls0\_pod =

lposet.new\_member(true, ls\_tuple\_id.hub\_pod());

lposet.put\_member\_name(ls0\_pod, "s0", true);

// Print the attribute tuples for the members.

cout << "v0 tuple: " << lposet.member\_dof\_map(lv0\_pod, false);

cout << endl;

cout << "v1 tuple: " << lposet.member\_dof\_map(lv1\_pod, false);

cout << endl;

cout << "s0 tuple: " << lposet.member\_dof\_map(ls0\_pod, false);

cout << endl;

// Make the segment cover the vertices.

lposet.new\_link(ls0\_pod, lv0\_pod);

lposet.new\_link(ls0\_pod, lv1\_pod);

// Top covers the segment.

lposet.new\_link(TOP\_INDEX, ls0\_pod);

// The vertices cover bottom.

lposet.new\_link(lv0\_pod, BOTTOM\_INDEX);

lposet.new\_link(lv1\_pod, lposet.bottom().index().pod());

// We're finished creating and linking jims.

lposet.end\_jim\_edit\_mode();

// Print the finished poset.

cout << lposet << endl;

// Write it to a file for later use.

storage\_agent lsa\_write("example22.hdf", sheaf\_file::READ\_WRITE);

lsa\_write.write\_entire(lns);

// Exit:

return 0;

}

### Base space member handles

There are a number of handle classes for members of base space posets: base\_space\_member and the several block handle classes.

#### Base\_space\_member

Base\_space\_member is the general interface to members of base\_space\_posets. It is a total\_poset\_member that knows about the attributes that all base space members have in common:

int db () const

The base space dimension.

const pod\_index\_type& type\_id () const

The cell type id of this. The id of the prototype of this in the "cell\_types" id space of the prototypes poset.

const char\* type\_name() const

The cell type name.

int refinement\_depth() const

The refinement depth.

#### Blocks

As we said before, a block is a collection of cells of the same type. In other words, a block is pretty much what we think of as a mesh, or at least a simple mesh. Some meshes require multiple blocks, perhaps of different kinds of cells.

More precisely, a collection of cells and the maximal dimension cells, which we refer to as "zones" or "elements", are all the same. Each maximal dimension cell may contain other kinds of cells in its downset. For instance, zones of any dimension typically contain vertices. The SheafSystem supports several different types of blocks, differentiated by the type of zones and how they are connected to each other:

Point blocks have zones are that isolated, disconnected 0-dimensional points. For class point\_block\_1d, the points can be indexed by a single index., class point\_block\_2d by a pair of indices (i, j) and class point\_block\_3d by a triple of indices (i, j, k)

Structured blocks are traditional i, (i,j) or (i,j,k) grids. The zones are 1, 2, or 3 dimensional boxes (line segments, quadrangles, or hexahedra, respectively) arranged in a regular 1, 2, or 3 index array and connected by their boundaries. Classes structured\_block\_1d, structured\_block\_2d, and structured\_block\_3d, respectively.

Zone-nodes blocks are finite element style meshes. The zones are line segments, triangles, quadrangles, tetrahedra, or hexahedra with explicit client-specified nodal connectivity. Class zone\_nodes\_block.

Unstructured blocks are more general finite element style meshes. The zones are copies of a client-specified zone template, connected with explicit client-specified nodal connectivity. The zone templates can, for instance, contain faces and or edges. The zone templates defined in the base space member prototypes poset are shown in Table 2.

Table 2: Standard prototypes for zone templates

|  |  |  |  |
| --- | --- | --- | --- |
| 0-cells | 1-cells | 2-cells | 3-cells |
|  | segement\_complex | triangle\_nodes | tetra\_nodes |
|  |  | triangle\_complex | tetra\_complex |
|  |  | quad\_nodes | hex\_nodes |
|  |  | quad\_complex | hex\_faces\_nodes |
|  |  |  | hex\_complex |

More general meshes, in fact any decomposition of a domain, can be represented using the sheaf system, but we will not discuss such meshes in this document.

The implementation of each block type is optimized for the type. Unstructured blocks can represent any of the other block types, but will not be as efficient.

### Subposets

#### D-cells subposets

### Traversing the graph

#### Connectivity iterators

#### Adjacency iterators

### Meshes

#### Mesh types

A mesh is a decomposition of some domain into a collection of smaller parts for the purpose of representing the dependence of one or more properties on position within the domain. In the sheaf system, a mesh is represented as a table, with a row for each of the basic parts in the decomposition. Each mesh typically consists of one or more "blocks". Each block is a collection of some number of "zones", all of the same type. The sheaf system supports several different types of blocks, differentiated by the type of zones and how they are connected to each other:

Point blocks have zones are that isolated, disconnected 0-dimensional points.

Structured blocks are traditional i, (i,j) or (i,j,k) grids. The zones are 1, 2, or 3 dimensional boxes (line segments, quadrangles, or hexahedra, respectively) arranged in a regular 1, 2, or 3 index array and connected by their boundaries.

Zone-nodes blocks are finite element style meshes. The zones are line segments, triangles, quadrangles, tetrahedra, or hexahedra with explicit client-specified nodal connectivity.

Unstructured blocks are more general finite element style meshes. The zones are copies of a client-specified zone template, connected with explicit client-specified nodal connectivity. The zone templates can, for instance, contain faces and or edges.

More general meshes, in fact any decomposition of a domain, can be represented using the sheaf system, but we will not discuss such meshes in this document.

The implementation of each mesh type is optimized for the type. Unstructured meshes can represent any of the other mesh types, but will not be as efficient.

#### Mesh index spaces

The sheaf system automatically generates and maintains the following id spaces for each mesh and block:

##### Mesh id space

Every member of a mesh table has a unique, persistent id in the hub id space of the mesh. As discussed above, scoped ids can be incremented, decremented, assigned, etc pretty much like ordinary integers. But one must start by getting an id that is in the desired scope. An id in the scope of the row id space can be obtained from the mesh object:

// Assuming base\_space\_poset lmesh has been initialized

// previously, get an id in mesh scope

scoped\_index lmesh\_id = lmesh.row\_scope\_id();

// Equivalent, but a little more efficient

scoped\_index lmesh\_id(lmesh.row\_scope\_id());

##### Mesh vertex and zone id spaces

Every mesh maintains a simple id space for the vertices in the mesh and a simple id space for the zones in the mesh. If the mesh contains cells of other dimensions, e.g. faces or edges, there are simple id spaces for them as well. An id in the scope of the id space for cells of dimension d can be obtained from the mesh object:

// Get an id in mesh vertex scope

scoped\_index lvertex\_id = lmesh.d\_cells\_id(0);

// Get an id in mesh zone scope, whatever dimension the mesh is

scoped\_index lzone\_id = lmesh.d\_cells\_id(lmesh.max\_db());

##### Block id space

Each block has a simple id space in which each member of the block, including the block itself, is given a sequential id, 0  block member id  block member count. This id space is referred to as the "local" id space for the block.

// Get an id in block local scope.

homogeneous\_block\* lblock;

scoped\_index llocal\_id = lblock->local\_id();

##### Block vertex and zone id spaces

Every point block, structured block, or zone\_nodes block maintains a simple id space for the vertices in the block and one for the zones in the block. Unstructured blocks do not currently support block vertex and zone id spaces.

// Get ids in block local vertex and zone scope.

homogeneous\_block\* lblock;

scoped\_index lvertex\_id = lblock->local\_vertex\_id();

scoped\_index lzone\_id = lblock->local\_zone\_id();

##### Block vertex and zone product id spaces

We said above that the block vertex and zone id spaces are simple, but since every product id space is also a simple id space, these id spaces may in fact be product spaces, if they were created that way. The vertex and zone id spaces for 2d and 3d structured blocks are always binary and ternary, respectively. A client can choose to create point blocks and zone\_nodes blocks with binary or ternary id spaces, see the examples below.

##### Client-defined id spaces

The client may create id spaces in addition to those defined above. We won't discuss these in this document.

#### Creating meshes

One creates a complete mesh in two steps: first one creates an empty mesh, then one creates one or more blocks within the mesh. The mesh object is referred to as the "host" for the blocks. Each block type has a static new\_host function for creating an appropriate host for blocks of the given type. Here's an example for blocks of type structured\_block\_2d:

// Step 1: create the host mesh.

base\_space\_poset\* lhost =

structured\_block\_2d::new\_host(lns, "2d mesh");

// Step 2: create a block within the host.

size\_type ni = 2, nj = 3

structured\_block\_2d lij\_grid(\*lhost, ni, nj, true);

// Step 3 (optional): give the block a name.

lij\_grid.put\_name("ij\_grid", true, true);

The third step, giving the block a name, is optional, but it is a good practice because it allows the client to refer to the block by path, which will be convenient when creating sections, see below.

#### Creating point blocks

Point blocks are collections of disconnected points, as mentioned above. A point block can be created with a product id space for the vertices: point\_block\_1d uses a simple id space but point\_block\_2d, and point\_block\_3d use binary and ternary id spaces, respectively.

// Create the mesh

base\_space\_poset\* lhost =

point\_block\_3d::new\_host(lns, "point mesh");

// Create a point block with a ternary vertex id space.

size\_type ni = 2, nj = 3, nk = 4;

point\_block\_3d lblock\_3d(\*lhost, ni, nj, nk, true);

For a point block, the vertices are also the zones, so the same id space applies to both vertices and zones.

Note that although we refer to point\_block\_3d, for instance, as having a "3d" id space, the indexing scheme itself has neither topological or geometric significance. Points are 0-dimensional objects. However, the indexing schemes can be useful when creating coordinates that assign the points positions in 1d, 2d, or 3d arrangements.

#### Creating structured blocks

Structured blocks are traditional multi-index array meshes. Structured\_block\_1d, structured\_block\_2d, and structured\_block\_3d correspond to i, (i,j), and (i,j,k) meshes, respectively. Structured\_block\_1d has zones that are line\_segments. The zones are arranged in a linear array and each segment except the last shares a vertex with the next segment, Similarly, structured\_block\_2d has zones that are quads arranged in a rectangular array. The zones implicitly share edges, although only the zones and the vertices are explicitly instantiated. Structured\_block\_3d has zones that are hexahedra arranged in a 3 dimensional array, implicitly sharing faces.

Structured blocks are always created with the product id space of the same "dimension" as the block. We've already seen the procedure for creating a structured\_block\_2d above; structured\_block\_1d and structured\_block\_3d are similar.

For structured blocks, the parameters specified in the constructors define the id space for zones. So in lij\_grid above, the zones are arranged in a 2 zones by 3 zones array. The vertices also have a binary id space, but the index bounds are each 1 greater; the vertices are arranged in a 3 by 4 array.

Unlike point blocks, the indexing scheme does have topological significance, since it must be compatible with how the zones share boundaries. For instance, zones i and i+1 in a structured\_block\_1d must share a vertex.

#### Creating zone-node blocks

Zone-node blocks are finite-element style meshes in which the zones are line segments, triangles, quads, tets, or hexs with explicit client-specified nodal connectivity. The connectivity is specified by creating a block\_connectivity object, then using it to create a block. As an example, let's create the following simple quad mesh:

Figure 1

// Create the mesh; since the zone\_nodes\_block class deals with

// any dimension, we have to tell it the maximum dimension of this mesh

const int lmax\_db = 2;

base\_space\_poset\* lhost =

zone\_nodes\_block::new\_host(lns, "2d zone\_nodes", lmax\_db);

// Create the connectivity object

const size\_type lct = 12;

pod\_index\_type lvertex\_ids[lct] = {0,1,2,3, 1,4,5,2, 2,5,6,7};

quad\_connectivty lquad\_conn(lvertex\_ids, lct);

// Create the block

zone\_nodes\_block lblock(\*lhost, lquad\_conn, true);

Zone\_nodes blocks are intended for representing irregular domains, but sometimes it's useful to create regular array arrangements, for instance when creating a simple test case. The connectivity classes provide a simple mechanism for doing this. For instance the following array arrangement of triangles can be easily created:

Figure 2

// Create a 2 x 2 array of triangles;

// note that the sizes refer to the

// number of edges in each direction.

triangle\_connectivity ltri\_conn(2, 2);

zone\_nodes\_block ltri\_block(\*lhost, ltri\_conn, true);

Now the vertices id space in ltri\_block is binary but the zones id space is simple.

#### Creating unstructured blocks

Unstructured blocks are finite element style meshes with more general types of zones than zone\_nodes blocks. For instance, using unstructured\_block one can create triangle meshes with explicitly instantiated edges, or hexahedral meshes with explicit faces and/or edges.

An unstructured block makes copies of a client-specified zone template and "glues" the copies together using client-specified nodal connectivity. A variety of zone templates are given in the poset base\_space\_prototypes, which is generated automatically as part of the fiber\_bundles\_namespace. Each template defines the cover relation graph for the zone, specifying which parts, e.g. faces and/or edges and/or vertices, are present and how they are contained in each other. The unstructured block builder copies the graph fragment for each zone in the mesh. The nodal connectivity provides enough information to identify how the copies are glued together, that is, how the copies share faces, edges, and vertices.

We can create the mesh of figure 1, but with the edges explicitly instantiated, using unstructured\_block:

// Create the mesh

const int lmax\_db = 2;

base\_space\_poset\* lhost =

unstructured\_block::new\_host(lns, "2d quad complex", lmax\_db);

// The zone template is specified by its path in the fiber bundles

// namespace. Quad with edges and vertices is "quad\_complex"

// Use the same nodal connectivity as the first zone\_nodes example.

poset\_path lpath("base\_space\_member\_prototypes", "quad\_complex");

unstructured\_block

lquad\_complex(\*lhost, lpath, lvertex\_ids, lct, true);

#### Multi-block meshes

As many blocks as one likes can be added to a given mesh. Just create the blocks using the same host. For instance, we can create the mesh in figure 3(a) as 2 structured blocks:

Figure 3

base\_space\_poset\* lhost =

structured\_block\_2d::new\_host(lns, "2d structured mesh");

structured\_block\_2d lblock\_1(\*lhost, 2, 1, true);

structured\_block\_2d lblock\_2(\*lhost, 1, 2, true);

Each structured block is disjoint from any other block, as depicted in figure 3(a). In order to join two blocks, as depicted in 3(b), one must explicitly glue them together, see "Gluing blocks together" below. Point blocks and zone node blocks behave the same as structured blocks in this regard, but unstructured blocks behave differently. If the connectivity is presented as implied by the numbering shown in figure 4(a), the blocks will automatically be glued together, as shown in 4(b).

Figure 4

#### Heterogeneous meshes

A single mesh can contain different types of blocks, as long as care is taken creating the mesh object. To create a heterogeneous mesh, use the new\_host method from the base class homogeneous mesh, and make sure the value provided for the xmax\_db argument is the dimension of the highest dimensional block in the mesh:

// Create a mesh to contain a 3d structured block and a 2d point block.

int const lmax\_db = 3;

base\_space\_poset\* lhost =

homogeneous\_mesh::new\_host(lns, "mixed\_mesh", lmax\_db);

// Add the blocks

structured\_block\_3d lblock\_3d(\*lhost, 2, 2, 2, true);

point\_block\_2d lblock\_2d(\*lhost, 3, 3, true);

#### Gluing blocks together

As mentioned above, blocks are disjoint by default. In particular, the vertices in one block are distinct from any other block; each vertex in each block initially gets a distinct id in the row id space of the table. Nevertheless, special features of the row id space allow us to force vertices, or any other mesh cells, to be shared between blocks. We call the process "gluing".

// Glue the blocks in Fig 3(a) together to form 3(b).

// First we need two ids in mesh vertex scope.

scoped\_index lvid\_a(lhost->vertices().id\_space());

scoped\_index lvid\_b(lhost->vertices().id\_space());

// Glue vertex 7 to vertex 5.

lvid\_a = 5;

lvid\_b = 7

lhost->glue(lvid\_b, lvid\_a);

// Glue vertex 6 to vertex 4.

lvid\_a = 4;

lvid\_b = 5

lhost->glue(lvid\_b, lvid\_a);

## Fiber spaces

### Fiber space schema

### Fiber space posets

### Fiber space members

### Fiber space member handles

### Subposets

### Traversing the graph

The classes in the fiber\_spaces cluster provide the various algebraic types used in theoretical physics to describe the properties of particles and systems. There are currently more than 35 such types, organized into an inheritance hierarchy based on their mathematical definitions. Some of the main branches in this hierarchy include:

* abstract vectors
* Euclidean vectors
* general tensors
* antisymmetric tensors
* symmetric tensors
* metric tensors
* Jacobians
* transformation groups

The sheaf schema for these types is defined in the "fiber\_space\_schema" table in the fiber\_bundle\_namespace.

See Appendix A for a discussion of the operations associated with these types.

### Using fiber types

Like meshes and all other sheaf types, a fiber space of a given type is represented by a table. To create instances of the type, we first have to create the table, then create the instance:

// Create a Euclidean vector space of dimension 3.

poset\* le3\_host = e3::new\_host(lns, "E3", true);

// Create some vectors in the space.

e3 i\_hat(le3\_host, true);

e3 j\_hat(le3\_host, true);

e3 k\_hat(le3\_host, true);

Class e3, like most of the fiber types, is a descendant of the abstract vector space class vd and hence has components.

// Get access.

lhost->get\_read\_write\_access(false);

// Set the components of i\_hat individually.

i\_hat.put\_component(0, 1.0, false);

i\_hat.put\_component(1, 0.0, false);

i\_hat.put\_component(2, 0.0, false);

// Set the components of j\_hat all at once.

j\_hat.put\_components(0.0, 1.0, 0.0);

Each fiber type is associated with an algebra; Euclidean vector algebra in the case of class e3. The algebraic operations are defined as namespace scope functions rather than member functions. Most operations are available in three forms: the "auto-allocated" form allocates its result on the heap and returns a pointer; the "pre-allocated" form requires a variable for the result as an argument; and the "self-allocated" form puts the result in its first argument. For instance:

// "Auto-allocated" add creates result on heap.

e3\* v1 = add(i\_hat, j\_hat);

// "Pre -allocated" cross product takes result as argument.

cross(i\_hat, \*v1, k\_hat);

When it makes sense, algebraic operations are overloaded onto C++ operators:

// Another way to compute the auto\_allocated sum.

e3& v1 = i\_hat + j\_hat;

// Or the "self-allocated" sum.

i\_hat += j\_hat;

// Release access.

lhost->release\_access();

The complete set of operations includes vector algebra, tensor algebra, symmetric algebra, and exterior algebra.

### Lite types

The fiber space types are most often used for short calculations, typically inside a loop where the fiber variable is just a temporary. For such purposes, the schema, access control, and persistence properties of the sheaf tables are unnecessary, inconvenient, and frequently inefficient. For this reason, every fiber type has an associated "lite" type, for instance e3 has e3\_lite. The lite types have all the same data members and algebraic operations as the full fiber types, but don't "live" in a table, don't have an explicit schema, and aren't access-controlled.

Lite types are intended to be easy and efficient to use as temporary variables. They can be created without a table:

// Create and initialize 2 lite vectors

e3\_lite ihat\_lite(1.0, 0.0, 0.0), j\_hat\_lite(0.0, 1.0, 0.0);

They support all the algebraic operations:

e3\_lite k\_hat = cross(i\_hat\_lite, j\_hat\_lite);

They can be easily converted to full fiber types:

i\_hat = ihat\_lite;

e3 v2(le3\_host, ihat\_lite, true);

### POD types

Both the full fiber types and the lite fiber types are C++ classes with virtual functions. As a result, the byte by byte layout of instances of these types is compiler defined and usually contains data in addition to the data members defined by the sheaf system. This makes it difficult to pass to any untyped, generic context, which includes the sheaf i/o subsystem as well as external C or FORTRAN functions. For this reason, each fiber type has a third associated type, the "plain old data" or "POD" type. (This nomenclature may seem informal to the point of being flippant, but it is in fact the official ISO C++ standard terminology!). For instance e3::row\_dofs\_type and e3\_lite::row\_dofs\_type both are typedefed to the pod type e3\_row\_dofs\_type.

The pod types satisfy the requirements of C++ POD types. They have no user-defined constructors or assignment operators and only public data members and member functions. They support data member access and conversion to and from full or lite types, but not algebraic operations.

## Sections

### Section space schema

### Section space posets

### Section space members

### Section space member handles

### Subposets

### Traversing the graph

#### Branch iterators

Informally, a section represents the dependence of some property on position within a domain, but we have to be more precise about what we mean by "position". We assume we have some set of patches that cover the domain and that each patch has a local coordinate system. So position in the domain means specifying which patch we're in and what local coordinates we're at within the patch. A section then represents a property as a function of patch and local coordinates. This is similar to the usual notion of "field" found in physics textbooks, but a field represents a property as a function of global coordinates; see the discussion of field objects below. The section formalism is more general than the field formalism in that the section formalism can be used when there is no single global coordinate system. The section formalism is also a good fit to the way properties are usually represented numerically.

### Section types and operations

The section type hierarchy repeats the fiber type hierarchy: for every type F in the fiber hierarchy there is a section type S with fiber of type F. Every fiber operation has a corresponding section operation. See Appendix B for additional discussion.

### Creating section spaces and sections

A section space is a table of sections, all of the same type over a given mesh. As with a mesh and the blocks it contains, a section space is referred to as the host for the sections it contains.

The procedure for creating a section space and sections is similar to creating a mesh and blocks, we have to create the section space before we can create the sections. But unlike a mesh, which has a predefined schema, the schema of a section space depends on the mesh the section space is based on. So before we can create a section space, we first have to create a schema for it. This makes creating a section a three step process: create the schema, create the section space, create the section.

We need to know three pieces of information to create the schema for a section space:

1. the base space, the mesh the section space is based on;
2. the fiber space schema, the type of the dependent variable; and
3. the representation type ("rep type"), the method used to discretize a section.

The base space must always be specified by the client, but a given section class may specify the fiber space and possibly the rep type. So each section class provides a static function make\_schema that takes the specific arguments needed to create a schema. For instance, the section class sec\_e2 corresponds to a 2D vector section. The fiber space (e2) is fixed for this section type, so the client need only specify the base space and representation.

A representation is identified by three further pieces of information:

1. the discretization subposet, the subset of cells in the mesh with which the section data ("degrees of freedom" or "dofs") are associated;
2. the evaluation subposet, the subset of cells with which the interpolation functions are associated; and
3. the evaluation method, the type of interpolation function.

The poset "sec\_rep\_descriptors" in the fiber bundles namespace provides a number of predefined rep types, with names derived from these three quantities. For instance, "vertex\_element\_dlinear" is a representation with the data at the vertices, interpolated over the zones (elements) using linear, bilinear, or trilinear interpolation, depending on the dimension of the mesh and the type of zone. Another example is "element\_element\_constant", for which the data is at the centers of the zones and is "interpolated" over the zones as a constant value. There are a number of other predefined rep types that address common cases, or the client can define specialized rep types.

So now we can create a schema:

// Step 1: Create a schema for a sec\_e2 section space

// on the ij grid defined above.

poset\_path lrep\_path("sec\_rep\_descriptors", "vertex\_element\_dlinear");

poset\_path lbase\_path = lij\_grid.path();

poset\_path lschema\_path =

sec\_e2::make\_schema(\*lns,

"e2\_on\_ij\_grid\_schema",

lrep\_path,

lbase\_path,

true);

Once we have a schema, we can create the section space. Each section class provides a static new\_host function for creating an appropriate section space:

// Step 2: create the section space.

sec\_rep\_space\* lsec\_e2\_host =

sec\_e2::new\_host(lns, "e2\_on\_ij\_grid", lschema\_path, true);

Finally, we can create a section:

// Step 3: create a section

sec\_e2 lsec\_e2(lsec\_e2\_host, 0, true);

### Accessing section values

#### Section id spaces

The data stored for each section are referred to as the "degrees of freedom" or "dofs" of the section and the collection of dofs is referred to as the "dof tuple". The schema defines what the degrees of freedom are. More specifically, an instance of the fiber is stored for each member of the discretization subposet in the base space and the fiber\_space\_schema defines the data that represents each instance of the fiber.

This means that there are 3 id spaces associated with the section dofs: the fiber dof id space, the discretization id space, and the section dof id space. The fiber dof id space is the row dof id space of the fiber schema. The discretization id space is the id space of the discretization subposet. The section dof id space is the Cartesian product of the discretization id space and the fiber dof id space. A specific example will help make this clearer.

For our sec\_e2 example, the fiber space is e2. The e2\_schema specifies 2 values of type double, the x and y components, so the fiber dof id space is {0, 1}.

The discretization subposet specified by the vertex\_element\_dlinear rep type is the vertices. Recalling that the ij\_grid was 3 x 4 vertices, the discretization id space is binary and it can be interpreted as the simple id space {0, 1, ... 11}.

The section dof id space is the Cartesian product discfiber so it can be viewed as the simple id space {0, 1, 2, ... 21}.

The interface for accessing the section dofs is based on these 3 id spaces, with an accessor function, a mutator function and an iterator for each id space.

#### Discretization id access

Typically the most convenient access is via discretization id. One can iterate over all the discretization "points" of a section and, since an instance of the fiber type is associated with each discretization id, one can get or put a fiber value:

// Assume sec\_e2 lsec1 and sec\_e2 lsec2 have been created

// previously and are members of the same section space.

// Create a temporary fiber value, a "lite" type.

sec\_e2::fiber\_type::volatile\_type lfiber;

// Get a discretization iterator.

// Note that the discretization id space, and hence the iterator,

// depends on the schema, not on the individual sections.

index\_space\_iterator\* ldisc\_itr =

lsec1.schema().discretization\_id\_space().iterator(true);

while(!ldisc\_itr->is\_done())

{

// Get the fiber value from the first section

// and put it in the second section.

lsec1.get\_fiber(ldisc\_itr->id(), lfiber, false);

lsec2.put\_fiber(ldisc\_itr->id(), lfiber, false);

// Increment the iterator.

ldisc\_itr->next();

}

delete ldisc\_itr;

If the discretization space is binary, as it is in this case, sometimes it's more convenient to use binary ids and conventional for loops:

double delx = 0.5, dely = 0.5;

product\_id\_space& lspace = lsec1.schema().discretization\_id\_space();

for(pod\_index\_type i=lspace.begin(0); i<lspace.end(0); ++i)

{

lfiber.put\_component(0, i\*delx);

for(pod\_index\_type j=lspace.begin(1); j<lspace.end(1); ++j)

{

lfiber.put\_component(1, j\*dely);

lsec2.put\_fiber(lspace.simple(i, j), lfiber);

}

}

#### Section dof id access

Occasionally it is more convenient to access the individual dofs. In that case, one can iterate over directly over the dof ids of a section and get or put an individual dof value:

// Create a temporary dof value.

sec\_e2::dof\_type ldof;

// Get a section dof iterator.

index\_space\_iterator\* ldof\_itr =

lsec1.schema().dof\_id\_space().iterator(true);

while(!ldof\_itr->is\_done())

{

// Get the dof value from the first section

// and put it in the second section.

lsec1.get\_dof(ldof\_itr->pod(), ldof, sizeof(sec\_e2::dof\_type));

lsec2.put\_dof(ldof\_itr->id(), ldof, sizeof(sec\_e2::dof\_type), false);

// Increment the iterator.

ldof\_itr.next();

}

delete ldof\_itr;

#### Fiber dof id access

One can also access the section values by fiber dof id. This access method is included mostly for mathematical completeness. It is rarely used in modern applications but can be useful in the context of legacy code oriented towards vector processor architectures.

Since the fiber dofs are usually the components of the fiber, the collection of section dofs with a given fiber dof id usually constitute a component of the section. Hence, this access method is called component access.

// Create a temporary section component.

// This is what makes this access method less useful than the others.

// Modern codes organize the section data x0, y0, x1, y1, ...

// not x0, x1, ..., y0, y1, ... so the section component is

// rarely an efficient or even convenient structure.

size\_type ldisc\_ct = lsec1.schema().discretization\_id\_space().size()

sec\_e2::dof\_type\* lcomp = new sec\_e2::dof\_type[ldisc\_ct];

size\_type lcomp\_size = ldisc\_ct\*sizeof(sec\_e2::dof\_type);

// Get a component dof iterator.

index\_space\_iterator\* lcomp\_itr =

lsec1.schema().fiber\_dof\_id\_space().iterator(true);

while(!lcomp\_itr->is\_done())

{

// Get the section component from the first section

// and put it in the second section.

lsec1.get\_component(lcomp\_itr->pod(), lcomp, lcomp\_size);

lsec2.put\_component(lcomp\_itr->id(), lcomp, lcomp\_size, false);

// Increment the iterator.

lcomp\_itr->next();

}

delete lcomp\_itr;

### Coordinate sections

The positions of the cells in a mesh are not an intrinsic feature of the mesh in the sheaf system. Positions are assigned to the cells by defining a coordinates section on the mesh. A mesh can have a single coordinates section or many coordinates sections. For instance, if the positions of the vertices change with time, then the coordinates at each time step can be stored as separate sections.

Any section which defines a one-to-one mapping of points in the base space to position can be used as a coordinates section, but the sheaf system provides two types of sections specifically intended for coordinates: uniform coordinates and general coordinates.

#### Creating uniform coordinates on structured blocks

Uniform coordinates can be used only on structured blocks. Uniform coordinates are defined by positions at the corners of the mesh and are interpolated uniformly (actually, d-linearly, where d is the dimension of the mesh) on the interior of the mesh. Assuming the positions are chosen to be a d-dimensional box, uniform coordinates make the mesh a "regular" grid - the vertices are evenly spaced and the zones are all the same size. Here's an example for the structured block named "ij\_grid" created above:

// Create a schema for a sec\_e2\_uniform section space.

poset\_path lbase\_path = lij\_grid.path();

poset\_path lschema\_path =

sec\_e2\_uniform::make\_schema(\*lns,

"e2\_uniform\_on\_ij\_grid\_schema",

lbase\_path,

true);

// Create the section space.

sec\_rep\_space\* lsec\_e2\_uniform\_host =

sec\_e2\_uniform::new\_host(lns, "e2\_on\_ijk\_grid", lschema\_path);

To create a uniform coordinates section, we need the positions of the corners of the domain:

// Create the lower bound (min x, min y) and

// upper bound (max x, max y).

block<sec\_vd\_dof\_type> llower\_bound("0.0 0.0");

block<sec\_vd\_dof\_type> lupper\_bound("2.0 2.0");

// Create a section

sec\_e2\_uniform lu\_coords(lsec\_e2\_uniform\_host,

llower\_bound,

lupper\_bound,

true);

lu\_coords.put\_name("ij\_grid\_uniform\_coordinates", true, true);

#### Creating general coordinates

General coordinates can be created on any type of block using the sec\_e1, sec\_e2, or sec\_e3 classes. For instance:

// Create a schema for a sec\_e2 section space.

poset\_path lbase\_path = lij\_grid.path();

poset\_path lschema\_path =

sec\_e2::make\_schema(\*lns,

"e2\_on\_ij\_grid\_schema",

lbase\_path,

true);

// Create the section space.

sec\_rep\_space\* lsec\_e2\_host =

sec\_e2::new\_host(lns, "e2\_on\_ij\_grid", lschema\_path);

// Create the section.

sec\_e2 lcoords(lsec\_e2\_host, 0, true);

lcoords.put\_name("ij\_grid\_coordinates", true, true);

The client has to explicitly set the dofs of the coordinates after creating the section. If the block vertex id space is a product space, it is often convenient to compute the dofs from a product index. For instance, in the current example we have a binary index:

// Set the dofs of the coordinates

// for a block with a binary id space.

double x0 = 0.0, delx = 0.5;

double y0 = 1.0, dely = 0.25;

product\_id\_space& lspace = lcoords.schema().discretization\_id\_space();

for(pod\_index\_type i=lspace.begin(0); i<lspace.end(0); ++i)

{

lfiber.put\_component(0, x0 + i\*delx);

for(pod\_index\_type j=lspace.begin(1); j<lspace.end(1); ++j)

{

pod\_index\_type ldisc\_id;

lspace.simple(i, j, ldisc\_id);

lfiber.put\_component(1, y0 + j\*dely);

lsec2.put\_fiber(ldisc\_id, lfiber);

}

}

In the case of more complicated mesh shapes, as is usually the case with zone-node and unstructured blocks, the block vertex id space is just a simple id space. The positions of the vertices were presumably determined by the mesher, at the same time the nodal connectivity was defined, and this position information can be used. For instance:

// Set the dofs of the coordinates

// for a block with only a simple id space

// using positions generated by mesher.

sec\_e2::dof\_type positions[][2]; // from mesher

index\_space\_iterator\* ldisc\_itr =

lcoords.schema().discretization\_id\_space().iterator(true);

while(!ldisc\_itr->is\_done())

{

lfiber.put\_components(positions[ldisc\_itr-pod()], 2);

lcoords.put\_fiber(ldisc\_itr->pod(), lfiber);

ldisc\_itr->next();

}

delete ldisc\_itr;

### Properties

The dependent variable (fiber) of a section can be any representable type. The fiber\_space cluster in the fiber bundles component of the sheaf system defines the types common in theoretical physics and the section\_space cluster defines section types for each of these fiber types. Creating sections of these types and accessing the dofs follows the patterns already described above for fiber type e2, only the fiber type is different.

### Multi-valued sections

Occasionally we need to represent a property that is multivalued. Consider a domain which consists of two parts ("patches") made of different materials. At the boundary between the two patches, density may have some value in the one material and a different value in the other material, making it effectively double-valued on the boundary. As another example, some objects, such as the surface of the Earth, can not be completely covered by a single coordinate system. In such cases the standard practice is to decompose the object into overlapping patches with a single-valued coordinate system on each patch.

In both these cases, the result is a set of ordinary single-valued sections, one on each patch. It is very useful to treat the result as a single section which is multi-valued where the patches overlap.

We call such multi-valued sections "multisections" and the sheaf system provides extensive support for them. In fact, multisections can be treated pretty much like ordinary sections, except for one difference: instead of having one set of fiber dofs for each discretization member, a multisection has one set of fiber dofs for each discretization member in each patch. This aspect is dealt with very naturally by defining the discretization id space of a multisection to be a sum id space.

More specifically, a multisection can be defined in any section space by specifying a subposet in the base space containing the patches on which the multisection is single valued. The intersection of the discretization subposet with each patch implies a set of "sub-subposets", each such sub-subposet containing the members of the discretization that are contained in the patch. The sheaf system automatically computes a simple id space for each of these sub-subposets and then creates their sum to use as the discretization id space for the multisection.

#### Creating a multisection

Creating a multisection just requires specifying the patches on which the section is single-valued:

// Assume we have a subposet containing the patches as members.

subposet lpatches;

sec\_e2 lmulti(lsec\_e2\_host, lpatches, true);

#### Accessing the dofs of a multisection

The dofs of a multisection can be accessed the same way as a regular section:

index\_space\_iterator\* ldisc\_itr =

lmulti.schema().discretization\_id\_space().iterator(true);

while(!ldisc\_itr->is\_done())

{

// Get the fiber value

lmulti.get\_fiber(ldisc\_itr->pod(), lfiber);

// Do something with the fiber

do\_something(lfiber);

// Put the modified value back in the multisection.

lmulti.put\_fiber(ldisc\_itr->pod(), lfiber);

// Increment the iterator.

ldisc\_itr->next();

}

delete ldisc\_itr;

Since the discretization id space is a sum, it may be more convenient to use binary ids and conventional for loops:

sum\_id\_space& lspace = lmulti.schema().discretization\_id\_space();

// Outer loop over patches; inner loop over disc ids within patch.

// Note that the begin and end of the disc id depends on the patch.

for(pod\_index\_type p=lspace.term\_id\_begin(); p<lspace.term\_id\_end(); ++p)

{

for(pod\_index\_type d=lspace.term\_begin(p); d<lspace.term\_end(p); ++d)

{

// Compute something that depends on patch and disc id

compute\_something(p, d, lfiber);

lmulti.put\_fiber(lspace.simple(p, d), lfiber);

}

}

## Fields

The classes in the fields cluster represent the usual physics notion of field, that is, some property as a function of global coordinates. The field abstraction is the natural context for a number of useful functions, most notably calculus. The fields cluster is still under construction at this time, so the functionality discussed here will only discuss a few of the features already implemented.

### Creating fields

We create a field from two sections, a coordinates section and a property section:

// Assume the scalar section lscalar has been created previously.

sec\_at0 lscalar;

// Create a scalar field

field\_at0 lscalar\_field(lcoords, lscalar, true);

### Evaluating the property at given coordinates

The defining function of the field object is to evaluate the property at given global coordinates. This is more complicated than it seems at first. The property is a section, a function of patch and local coordinates. To evaluate it, we need to know what patch and local coordinates the global coordinates value corresponds to, in other words, we have to invert the global coordinate section. The task is even more complicated if the coordinates and the property aren't defined on the same discretization and evaluation subposets, for example if the coordinate representation is uniform and the property representation is vextex\_element\_dlinear. The property\_at\_coordinates method takes care of all this complexity:

// Create a global coordinates value

// and a buffer for the result.

sec\_vd\_dof\_type lglobal[2] = {0.5, -0.5};

sec\_vd\_value\_type result;

// Evaluate the property at the given global coordinates.

lscalar\_field.property\_at\_coordinates(lglobal, 2, result, 1);

### Creating a property as a function of the coordinates

Its quite common to want to set the property to some given function of the coordinates. If the coordinates and the property sections both use the same discretization, the task is simple - we just iterate over the discretization id space, get the coordinate dofs, compute the given function and put the property dofs. But if the property and coordinates are not on the same discretization, the task is substantially more complicated. For each property discretization point, we have to find the coordinate evaluation member that contains the point, gather the coordinate dofs for that evaluation member, evaluate the coordinate section at the property discretization point, then compute the property dofs from the coordinate dofs. The put\_property\_dofs function takes care of all this for us. All the client has to provide is a simple function that does the last step - calculates the property dofs given the coordinates:

// First we need to define a function to compute

// the property dofs as the desired function of the coordinates.

// As an example, assume the property is scalar and set it to

// the distance from the coordinate origin.

void

property\_dof\_function\_example(block<sec\_vd\_value\_type>& xglobal\_coords,

block<sec\_vd\_dof\_type>& xproperty\_dofs)

{

sec\_vd\_value\_type dist = 0.0;

for(int i= 0; i<xglobal\_coords.ct(); ++i)

{

dist += xglobal\_coords[i]\* xglobal\_coords[i];

}

dist = sqrt(dist);

xproperty\_dofs[0] = dist;

return;

}

// Now use the function to set all the property dofs.

lscalar\_field.put\_property\_dofs(property\_dof\_function\_example, true);

### Pushing a property from one mesh to another

One of the most useful features of the field abstraction is that it provides enough context to support moving, or "pushing", a property from one mesh to another. As with property\_at\_coordinates and put\_property\_dofs, this task is considerably more complicated than it appears at first. The push\_to function takes care of the complexity and makes it easy:

// Assume lscalar\_field1 and lscalar\_field2 have been

// previously created on different meshes.

field\_vd lscalar\_field1, lscalar\_field2;

// Move the property of lscalar\_field1 to

// the mesh of lscalar\_field2.

lscalar\_field1.push\_to(lscalar\_field2, true);

Alternatively, the right shift operator is overloaded to invoke the push\_to\_function.

// Move the property of lscalar\_field1 to

// the mesh of lscalar\_field2.

lscalar\_field1 >> lscalar\_field2;

1. Concurrency control examples

The access control mechanism is a work in progress. The control mechanism itself is complete and is implemented both for multiple threads using pthreads and for single threads. When the library is compiled with threads enabled and a client requests read access and another client already has write access, or vice versa, the request blocks until the other client releases the conflicting access. When the library is compiled with threads disabled, requests do not block, they return immediately. The library is currently delivered with threads disabled because the use of threads and concurrency in the library is only partially implemented and not tested. The access control mechanism is disabled by default but can be enabled by the programmer. These examples demonstrate use of the manual and auto-access mechanisms.

* 1. Example A1: manual access control

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide ExampleA1:" << endl;

// Enable concurrency control; must be called

// before any other library call.

read\_write\_monitor::enable\_access\_control();

// Create a standard sheaves namespace.

sheaves\_namespace\* lns = new sheaves\_namespace("ExampleA1");

// Write its name to cout.

// Requires read access to the namespace.

// Be polite, request access.

// If threads are enabled and another thread has

// read-write access, execution will block until it

// releases access. Otherwise, the request will succeed

// immediately.

// You can nest requests as deep as you want, or at least

// until the integer depth counter overflows.

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->get\_read\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->get\_read\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

// Invoke the operation.

cout << lns->name() << endl;

// Be proper, release access so this thread

// or another can get write access.

// Have to match every request with a release.

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->release\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->release\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

// Delete the namespace, requires read-write access.

// Be polite, request access. If threads are enabled

// and another thread has either read or read-write

// access, execution will block until it releases access.

// Otherwise, the request will succeed immediately.

// This client must not already have read-only access,

// see precondition for details.

lns->get\_read\_write\_access(false);

// Invoke the operation.

delete lns;

// Deletion is the only case where the client

// cannot be proper and release access.

// Create another namespace.

lns = new sheaves\_namespace("Example3B");

// Invoking a function that requires access

// without first getting access violates the

// precondition of the function.

// The following will throw an exception and abort.

cout << lns->name() << endl;

return 0;

}

If you compile and run example A1, the output is:

SheafSystemProgrammersGuide ExampleA1:

request depth 0

request depth 1

request depth 2

Example3A

request depth 2

request depth 1

request depth 0

terminate called after throwing an instance of 'std::logic\_error'

what(): 'is\_external() ? name\_space()->state\_is\_read\_accessible() : state\_is\_read\_accessible()' in file poset\_state\_handle.cc at line 1178

Abort

* 1. Example A2: auto-access control

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide ExampleA2:" << endl;

// Enable concurrency control; must be called

// before any other library call.

read\_write\_monitor::enable\_access\_control();

// Create a standard sheaves namespace.

sheaves\_namespace\* lns = new sheaves\_namespace("ExampleA2");

// Write its name to cout.

// Requires read access to the namespace.

// Invoke the auto-access version of the operation with

// auto-access set to true.

// Operation will request and release access as needed.

cout << lns->name(true) << endl;

return 0;

}

If you compile and run example A2, the output is:

SheafSystemProgrammersGuide ExampleA2:

ExampleA2

1. Join equivalent members and lexicographic ordering

Is it legal to have a jrm with a single member in its lower cover? The answer is no, in an ordinary finite distributive lattice as defined in the text books, but yes, in a SheafSystem lattice. In the mathematical view, the lattice is fully instantiated, the order relation fully enumerated, and a member is a jim if and only if it has a single member in its lower cover. So c1 couldn't be a jrm. In this view, the join operator is a query that finds the member which is the least upper bound of the joinands. The Birkhoff representation theorem is a consequence of the order relation and it, in turn, implies two additional facts. First, for lattice members p and p', the set of jims in the downset of p is included in the set of jims of p' if and only if p  p'. Second, the jims in the downset of a join is the union of the jims of the joinands.

But on the computer, we can't afford the memory to fully instantiate all the members of the lattice and enumerate the order relation. So the SheafSystem reverses the mathematical argument. We initially instantiate only the poset of join irreducible members and the covering relationships between them. That is, a member is a jim because we specify that it is when we create it, not because of some property of the cover or order relation. The join operator isn't a query, it's a constructor that creates a jrm and places it in the cover relation so that the set of jims in its down set is the union of the jims of the joinands. The Birkhoff representation theorem is satisfied by construction and the order relation is derived form it: p  p' if and only if the set of jims in the down set of p is included in the jims of p'.

But what if we join the same set of joinands twice? We can interpret the second operation in two ways. Either the join operator finds the existing join and returns it as the result, or it constructs a second member, with the same jims in its downset as the first join. The SheafSystem supports both approaches, but in the latter case, where should the second join be placed in the order relation? The set of jims of the second join is equal to the set of jims of the first, so the order relation says they are equal, but they are not the same object. The SheafSystem breaks the tie by extending its definition of the order relation to what is called a lexicographic order. A lexicographic order is a generalization of ordinary dictionary order. To place words in order, first we sort on the first letter. If two words have the same first letter, we sort on the second letter, and so on. The SheafSystem uses a lexicographic order in which the first letter is the set of jims in the down set and the second letter is the order of creation. So when we construct the second join, it has the same set of jims but it was constructed after the first join, so it is greater than the first join. The second join is thus linked immediately above the first join. A third copy would be linked above the second copy, and so on.

1. Fiber Algebra

As described above, the classes in the fiber\_spaces cluster provide the various algebraic types used in theoretical physics to describe the properties of particles and systems. Each of these algebraic types has an associated set of operations, an algebra. In this appendix we summarize the operations associated with each type. See the reference documentation for a complete description of the operations and the numerous signatures supported for each operation.

* 1. Abstract vectors (vector algebra)

Class vd and its descendants support the operations of vector algebra (linear algebra). The main operations are:

add add one vector to another.

subtract subtract one vector from another.

multiply multiply a vector by a scalar.

divide divide a vector by a scalar.

min find the minimum component of a vector.

max find the maximum component of a vector.

contract contract a vector with a covector.

* 1. Euclidean vectors (Euclidean vector algebra)

dot the Euclidean scalar product of two vectors.

length the length of a vector.

put\_length sets the length of a vector.

normalize sets the length of a vector to 1.

cross the vector ("cross") product of an e3 vector with another.

* 1. General tensors

Class tp and its descendants support general tensor algebra. The main operations are:

tensor the tensor product of one tensor with another.

alt the antisymmetric ("alternating") part of a tensor.

sym the symmetric part of a tensor.

contract contract a tensor on one contravariant and one covariant index.

* 1. Antisymmetric tensors (exterior algebra)

Class atp and its descendants support exterior algebra. the main operations are:

wedge the exterior ("wedge") product of two antisymmetric tensors.

hook the interior ("hook") product of an antisymmetric tensor and a vector.

star the Hodge star ("dual") of an antisymmetric tensor.

* 1. Symmetric tensors (symmetric algebra)

symmetric\_product the symmetric product of one symmetric tensor with another.

trace the trace of a degree 2 symmetric tensor.

determinant the determinant of a degree 2 symmetric tensor.

to\_principal\_axes diagonalizes a degree 2 symmetric tensor

* 1. Metric tensors

raise make a given index of a tensor contravariant.

lower make a given index of a tensor covariant

* 1. Jacobians

push push a vector from the domain to the range of a Jacobian

pull pull a covector from the range to the domain of a Jacobian

* 1. Transformation groups

inverse the inverse of a linear transformation

transform\_basis\_by transform the basis and components of a tensor

* 1. Multiple signatures

Each of the above operations is represented by a family of functions implementing multiple signatures for the operation. Typically each operation is provided for both regular fiber types and the associated lite fiber types. In addition, auto-allocated, pre-allocated and self-allocated variants are provided. Finally, whenever it makes sense, operator variants are provided. As an example, the full set of functions for the add operation is as follows:

///

/// x0 add x1 (auto-allocated version for lite types).

///

template <typename T> T\* add(const T& x0, const T& x1);

///

/// x0 add x1 (pre-allocated version for regular types).

///

void add(const vd& x0, const vd& x1, vd& xresult, bool xauto\_access);

///

/// x0 add x1 (pre-allocated version for lite types).

///

void add(const vd\_lite& x0, const vd\_lite& x1, vd\_lite& xresult);

///

/// x0 add\_equal x1 (self-allocated version for regular types);

/// synonym for add(xresult, xother, xresult, xauto\_access).

///

void add\_equal(vd& xresult, const vd& xother, bool xauto\_access);

///

/// x0 add\_equal x1 (self-allocated version for lite types).

///

template <typename T> void add\_equal(T& xresult, const T& xother);

///

/// x0 + x1 (auto-allocated for lite types);

/// synonym for add(x0, x1).

///

template <typename T> T\* operator+(const T& x0, const T& x1);

///

/// x0 += x1 (self-allocated for lite types);

/// synonym for add\_equal(x0, x1).

///

template <typename T> T& operator+=(T& xresult, const T& xother);

The full fiber algebra consists of all the various signatures for all the various operations and thus contains approximately100 functions.

1. Section algebra

The section type hierarchy repeats the fiber type hierarchy: for every type F in the fiber hierarchy there is a section type S with fiber of type F. Every fiber operation has a corresponding section operation. Sections also support additional operations, not defined for fibers. The most important of these are the real-valued functions of scalar sections. A scalar section is essentially a map from the base space to reals and we can compose this with any map from reals to reals to form another map from the base space to reals, i.e. another scalar section. We define such an operation for each math library function defined by the C++ standard. We summarize these in the following:

fabs absolute value of a scalar section.

ceil ceiling of a scalar section.

floor floor of a scalar section.

sqrt square root of a scalar section.

pow power of a scalar section.

cos cosine of a scalar section.

sin sine of a scalar section.

tan tangent of a scalar section.

acos arc cosine of a scalar section.

asin arc sine of a scalar section.

atan arc tangent of a scalar section.

atan2 arc tangent of a scalar section.

cosh hyperbolic cosine of a scalar section.

sinh hyperbolic sine of a scalar section.

tanh hyperbolic tangent of a scalar section.

exp exponential of a scalar section.

log log base e of a scalar section.

log10 log base 10 of a scalar section.

modf integer and fractional parts of a scalar section.

frexp fractional and exponent parts of a scalar section.

fmod remainder of scalar section.

ldexp scalar section times a power of 2.