It’s Scopes All the Way Down

Document Number: P3955R1
Date: 2026-08-09
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG, SG1

Abstract

This paper lays out an approach to achieving the same effect as regular, synchronous scopes in the asynchronous domain (i.e. `std::execution` [1]), including asynchronous construction and destruction of objects.

Background

Even C has scopes. Unlike C++, however, nothing (except allocating and deallocating stack space) can be accomplished on their boundaries.

The power of C++ constructors and destructors, referred to as “RAII” (even when resources are not being acquired) is the ability to inject arbitrary code into the rising and falling edges of a scope. This not only trivializes tasks which are error prone in languages such as C (allocating and freeing memory, for example) but also allows for a wide range of creative and heterodox applications.

Unfortunately C++ is a fundamentally synchronous language. Entering and leaving a scope, and therefore the power of RAII, is only available synchronously. Even where C++ attempts to be an asynchronous language it does not extend the power of its scopes thereto, saying instead (§11.4.5.1 [class.ctor.general]):

“A constructor shall not be a coroutine.”

And (§11.4.7 [class.dtor]):

“A destructor shall not be a coroutine.”

But C++ is more than a language, it is also a library. So much so that C++ refuses fundamental features of other languages, leaving them instead to the library. Sum and product types, for example. An asynchronous simulacrum of itself, as another ([2] at §3):

“It is becoming apparent that all the sender/receiver features are language features being implemented in library.”

Out of the box in C++26 this simulacrum, `std::execution`, featured an upgraded version of the C++ language construct of a function (ibid.):

"Sender/Receiver itself is the implementation of an `async-function`*."*

Unlike the built in function call protocol the asynchronous operations of `std::execution` are:

- Fundamentally asynchronous,
- Integrate cancellation as a first-class completion [3], and
- Offer more flexible representations of completions, including homogeneous completions for both the value and error channels, and value completions which send multiple types (i.e. explicit sum and product types are not needed in the completion signatures of a sender)

This however just projects functions into the asynchronous domain. C has functions, but does not have RAII. `malloc` and `free` are functions, `unique_ptr` is something else: An object.

To evolve its asynchronous ecosystem and achieve parity with its synchronous ecosystem C++ needs not only asynchronous functions, but also asynchronous objects [2].

Discussion

Prior Art

Kirk Shoop has previously explored this space [2]. With his permission the author of this paper is continuing that exploration.

What Is an "Asynchronous Object?"

A synchronous object is an object whose lifetime begins via a synchronous function (i.e. a constructor) and ends via a synchronous function (i.e. a destructor). Similarly ([2] at §1.1):

"An `async-object` *is an object [...] that has* `async-function`*s to construct and destruct the object."*

Put differently: An asynchronous object is an object whose constructor and destructor are asynchronous.

Why Are Asynchronous Objects Needed?

With the idea of an asynchronous object described the next question is: Why is such a concept needed or useful? Are asynchronous constructors and destructors really needed? Are they something users of C++ would ever require or reach for?

Consider an object which represents and manages a TCP connection to a server. If it has an invariant that it is connected (and that it is an error for it not to be) then it cannot be said to be fully constructed until the asynchronous process of connecting (and perhaps resolving a DNS address or trying other endpoints) has completed. Two phase init could clearly be used here, but that is widely regarded as an antipattern in general (the Google style guide, for example, has historically been much-maligned for featuring/requiring it). Therefore an asynchronous constructor (or other factory) is needed.

But the above-described object still has a synchronous destructor: The POSIX `close` system call on the file descriptor representing the connected socket. However the existence of io_uring, a fundamentally asynchronous way to make Linux system calls, muddies these waters as it features an asynchronous equivalent even of `close`: `IORING_OP_CLOSE`.

POSIX system calls and io_uring are beyond the purview of the standard, however, and for justification of asynchronous destructors we need not look that far. C++26 actually contains such a justification: Async scopes [4]. Consider the functionality of the synchronous destructor of `execution::simple_counting_scope` (§33.14.2.2.2 [exec.simple.counting.ctor]):

"If `state` *is not one of* `joined`*,* `unused`*, or* `unused-and-closed`*, invokes* `terminate`*. Otherwise, has no effects."*

The scope is transitioned to the *`joined`* state by obtaining a sender via the scope's `join` member function, connecting it, starting the resulting operation state, and allowing that asynchronous operation to run to completion. That is to say: `execution::simple_counting_scope` has manual, asynchronous clean up which must be performed before the object can be allowed to go out of scope. Clean up is the raison d'être of RAII and one of C++'s greatest strengths, but because C++ doesn't have asynchronous RAII the clean up story for this type is more reminiscent of C: Make sure you follow the right idiom, make sure you don't forget, make sure whoever is maintaining the code after or with you does the same.

A narrow solution to the specific problem above has been proposed: `execution::let_async_scope` [5] (this will be returned to later in this paper), but we can do better. We already have the principle of RAII which addresses this class of problem, we just need to project it into the asynchronous domain.

Prior Design

The prior design (from [2]) centered around types which modelled the `execution::async_object` and `::async_object_constructible_from` concepts. Note that a type which modelled the async object concept was not itself the object which would be created or interacted with, but rather a factory which could be used to produce an object. One can think of an async object in this design as a partially-curried, but asynchronous, version of a constructor, in the same way a sender is a fully curried, but asynchronous, version of a function.

An async object was required to be shaped in the following way (id. at §4.2.0.1):

```
struct async_object {
  using object = /* ... */;
  using handle = /* ... */;
  using storage = /* ... */;
  execution::sender auto async_destruct(storage&) const noexcept;
  template<typename... Args>
  execution::sender auto async_construct(storage&, Args&&...) const;
};
```

With:

- `object` being the actual type of the object to asynchronous construct and destroy,
- `handle` being the type through which above-mentioned object will be accessed (i.e. it is the result of async construction), and
- `storage` being the type which provides storage for the object

The utility `execution::make_packaged_async_object` was proposed (id. at §4.2.0.6) to curry arguments into an async object thereby rendering it “default async-constructible” (i.e. having an `async_construct` member function which accepts no trailing arguments). This meant that the above-described async object design followed the model of iterative currying already embodied by senders and receivers wherein:

1. Arguments are applied into a sender by a sender factory, and then
2. A receiver is applied into an operation state by `execution::connect`

As follows:

1. Arguments are applied into an async object to yield an async object whose `async_construct` member function is unary (i.e. requires only storage),
2. Storage is applied into an async object to yield a sender, and then
3. A receiver is applied into an operation state by `execution::connect`

This applicative model is powerful because it allows elegant separation of concerns, as evidenced by the generic algorithms of `std::execution`.

Issues With the Prior Design

Reference Qualification

The most serious issue with the previous design is that it requires that `async_construct` and `async_destruct` be invocable on const lvalue async objects. This means that either:

- Curried arguments must be copyable into the senders yielded by the above-mentioned member functions (so they can be propagated from a const object), or
- The async object must remain within its lifetime until the completion of asynchronous construction and destruction thereof

Even the latter of the two above possibilities, which has the fewest performance implications, has a problem. Even if the async object remains within its lifetime, thereby allowing the constructor and destructor senders (and their resulting asynchronous operations) to refer to the contents thereof there's still no way to use those contents consumptively (e.g. by moving therefrom) because the capture occurred from a const-qualified object.

It might initially seem that the design can easily be extended to rectify the above: Simply cause the `async_object` concept to check for invocability of `async_construct` and `async_destruct` with the const-ness and reference-ness of whatever type is provided (for example `execution::sender_to` does this with connectability). However the protocol consists of two member function invocations: Both `async_construct` and `async_destruct` must be called for the corresponding object to be created and destroyed. This means that if `async_construct` is called on an rvalue-qualified object there must be a subsequent invocation of `async_destruct` which might then arguably constitute a use after move.

Curried or Not?

Because of the unfixed arity of `async_construct` (i.e. it may have trailing arguments after the reference to the storage) the previous design provides two concepts: `execution::async_object` and `::async_object_constructible_from` with only the latter checking the entire protocol. I.e. it is possible to satisfy `execution::async_object` and have no `async_construct` member function at all, because it's impossible to check the `async_construct` member function without knowing which arguments you expect to invoke it with (in some sense this is similar to `execution::sender` and `::sender_in`).

An advantage of the above-described design is that it mirrors regular, synchronous C++ objects. Such objects can have many constructors, callable in many different ways. But there's an important difference between regular, synchronous objects and async objects: An async object

is not the object it constructs. The async object represents an indirection, hence the need for the `object` nested type.

The above formulation creates blurred categories: An arbitrary async object is an async object, but so is the async object created by `execution::packaged_async_object`. Despite being in the same category they don't support the same operations, and therefore can't be interacted with generically.

This creates a question with respect to the single responsibility principle: What is the responsibility of the caller of `async_construct`? Is it to manage the lifetime of a certain object, or is it to select and apply the arguments which will select the object to be constructed?

`std::execution` itself already contains an example of an alternate design: The arguments to the asynchronous function call represented by a sender are provided to a sender factory which generates a sender. This sender is isomorphic to a fully-curried synchronous function. There is a single responsibility in selecting the arguments. Elsewhere, in another domain of responsibility, that sender is connected and started. This decomposition is foundational to the generic power of `std::execution` and its asynchronous algorithms and should be reflected in the design of asynchronous RAII.

Object Primacy

Constructors and destructors are fundamentally associated with objects (since they respectively start and end the lifetimes thereof). Objects have lifetimes which are associated with scopes (note that a scope may be lexical or more abstract). Therefore constructors and destructors are ways to run code when a scope is entered or exited, however they're still coupled to an object. This friction can be seen in the idiom of the "scope guard" [6] wherein an object is synthesized solely to perform some action when its lifetime ends. In some sense this "scope guard" object is not a "real" object and is instead an abuse of objects to instrument the exit from a scope, since C++ does not have a native way to access such functionality.

The prior design of async RAII too engages in the above-described object primacy: Async objects in that design are always associated with storage thereby marrying the running of code when entering and exiting a scope to the provisioning of storage and the creation of an object.

Given how common the scope guard idiom is in C++ the design of async RAII should not cling to the above-described limitation of C++ RAII in the same way that `std::execution` didn't cling to the limitations on the structure of synchronous function return (i.e. unlike regular, synchronous functions the asynchronous functions can report completion in many ways with each yielding potentially many values).

New Design

Scopes

The fundamental building block of the design proposed by this paper is a scope. As discussed above such scopes are not coupled to storage or objects. Entering a scope requires some action be performed, and leaving the scope also requires some action be performed. Both of these actions are represented by senders and can therefore be performed asynchronously.

Unlike the prior design wherein a caller had immediate access to construction (i.e. entering a scope) and destruction (i.e. leaving a scope) by calling the appropriate member functions of an async object in this design the “enter scope sender” (which performs the action necessary to enter the scope) yields the “exit scope sender” (which performs the action necessary to leave the scope) upon completion, i.e. the enter scope sender is a higher order sender. This avoids the problem of rvalue qualification because the enter scope sender is consumed to form the enter scope operation, which then yields another entity, the exit scope sender, which may then be consumed later in turn when the time comes to leave the scope.

Scope Algorithms

The fundamental scope algorithm is `within`. It establishes a scope by running an enter scope sender. Thereafter it allows a child operation to run (in that scope since the exit scope sender has not yet run), and then before completing the overall operation runs the exit scope sender yielded by the enter scope sender.

Enter scope senders can be composed in one of two ways: In sequence, or in series. These two patterns are reified by two scope sender adaptors:

- `enter_scopes` combines N enter scope senders into an enter scope sender that enters all N scopes in parallel and then yields an exit scope sender that exits all N scopes in parallel
- `nest_scopes` combines N enter scope senders into an enter scope sender that enters each scope in order and then yields an exit scope sender that exits each scope in reverse order

Note that if any enter scope sender’s operation fails the appropriate exit scope sender’s operation will be synthesized and allowed to run before the overall operation fails.

Objects

Objects combine a scope with storage. Entering the scope constructs an object in the storage, and exiting the scope destroys that object. An async object now has the following interface:

```
struct async_object {
  using type = /* ... */;
  execution::enter_scope_sender auto operator()(type*);
};
```

Note that since this entity has a single basis operation (i.e. invocation) rvalue qualification of invocation is trivially supportable.

The above design also reflects and extends the currying model of `std::execution`. The workflow to run an asynchronous operation consists of:

1. Currying arguments into the sender,
2. Connecting the sender to a receiver, and then
3. Starting the asynchronous operation

Whereas synthesis of an asynchronously-constructed and -destroyed object consists of:

1. Currying constructor arguments into the async object,
2. Currying the storage pointer into the enter scope sender,
3. Connecting the sender to a receiver, and then
4. Starting the asynchronous operation

With only one responsibility present at each step (selection of arguments, followed by selection of storage, followed by selection of continuation).

Object Algorithms

The fundamental async object algorithm is `lifetime`. It accepts N async objects and provides storage for all of them. Once the asynchronous operation is underway it constructs all objects in parallel and then provides references thereto to a user-provided invocable which synthesizes a sender in response (like `execution::let_value` et al.). This sender is connected and started and once it completes the exit scope senders are connected and started to destroy all async objects after which the operation ends.

Commentary on Naming

Maybe we should've called a "scope" [4] a "nursery." The author is not particularly attached to the names in this paper and imagines they will be changed.

Note that none of the names in this paper, unlike the previous one [2], feature the `async_` prefix. This is because the author believes that entities in the `std::execution` namespace are presumptively async [7].

Binary Concepts

The fact that a type satisfies the `sender` concept alone is not sufficient to indicate whether or not it may be used in a given situation. Asynchronous operations are connected within a certain environment. This justifies the `sender_in` concept which ascertains whether, in addition to being a sender generally, a certain type is a sender within a certain environment.

The above formulation might seem to indicate that `sender_in` is a binary concept accepting the sender type and the environment type. However this is not the case. Just because some senders are not senders in certain environments (i.e. `sender<Sndr>` is `true` while `sender_in<Sndr, Env>` is `false` for some `Env`) does not preclude the existence of senders which are senders in any environment (i.e. `sender_in<Sndr, Env>` is `true` for every `Env`). The latter kind of sender is said to be “non-dependent” (because its properties do not depend on the environment) and to support such senders the `sender_in` concept is variadic (i.e. `sender_in<Sndr>` is valid) [20].

The design proposed by this paper adds three `_in` concepts which are paired with unsuffixed versions:

- `exit_scope_sender_in` (and the unsuffixed `exit_scope_sender`)
- `enter_scope_sender_in` (and the unsuffixed `enter_scope_sender`)
- `object_in` (and the unsuffixed `object`)

Unlike `sender_in`, however, the above concepts are binary. The idea of a non-dependent exit scope sender, for example, is not supported (or, at least, is not surfaced by the concepts proposed herein).

The reason for this is that, as discussed above, use of exit scope senders must not result in failure. Decay-copying them must not fail. Move constructing them must not fail. Connecting them must not fail. Their resulting asynchronous operation must not have error or stopped completion signatures.

For an exit scope sender which is not a dependent sender most of the above determinations can be made without the context of an environment. Decay-copy and move do not depend on the environment and therefore their throwingness can trivially be determined without an environment. Non-dependent senders do not require the context of an environment to determine their completion signatures and therefore checking to ensure there are no error or stopped completion signatures can be done without the context of an environment. Determination of the throwingness of `connect` alone is problematic.

The reason for the above is the way in which it is determined whether or not `connect` throws: By interrogating a `connect` expression via the `noexcept` operator [8]. This historically required a receiver, rather than an environment, but that was fixed late in the C++26 cycle (id.).

At first glance the above seems as though it requires only `exit_scope_sender_in` to be binary (rather than variadic) since the above discussed centers on exit scope senders. However consider that:

- To be an enter scope sender the sole value completion must transmit an exit scope sender, and that therefore checking if something is an enter scope sender checks to see if something else is an exit scope sender
- To be an async object invocation must return an enter scope sender (see preceding bullet)

An alternative, suggested by Eric Niebler, which would allow the above-mentioned concepts to be variadic (rather than binary) would be for the concepts to mandate that `connect` does not throw, rather than constrained based on this. This would mean that evaluating, for example, `exit_scope_sender_in` with two arguments where connecting an instance of the first argument in the environment given by the second argument would yield a hard compilation error rather than the concept simply evaluating to `false`.

The above formulation not only allows for exit scope sender-ness to be dependent, it also reduces the rate at which confusing compilation errors occur. A user might believe they've written an exit scope sender, but have neglected `noexcept` on the `connect` operation (or accidentally done something which caused said operation to be computed as `noexcept(false)`). This leads to unexpected and often-misleading compile errors. Using mandates, rather than constrains, would cause a hard compilation error which would indicate the problem precisely.

For the time being the original, binary design has been preserved in this paper. Taking the approach described above is left as a suggested/desired poll (see section of suggested/desired polls below).

Proposed Design

Concepts

`exit_scope_sender<Sender>`

Determines whether a sender has the properties necessary to be an exit scope sender without the context of an environment. Note this concept is described first because it's at the bottom of the hierarchy: Enter scope senders transmit an exit scope sender, and therefore that concept needs to check this one. Objects yield enter scope senders, thereby rooting this concept as the most foundational.

An exit scope sender must be a sender, and must not throw when decay-copied or moved. Other properties require an environment.

`exit_scope_sender_in<Sender, Env>`

Extends `exit_scope_sender` by checking properties that require an environment. Additionally checks if the sender can be nothrow connected in the environment [8], and that it completes in exactly one way: `set_value_t()` (i.e. checks that it always succeeds and doesn't send any "result datums").

`enter_scope_sender<Sender>`

Equivalent to `execution::sender<Sender>`. Without an environment nothing further can be gleaned.

`enter_scope_sender_in<Sender, Env>`

Extends `enter_scope_sender` by checking properties that require an environment (which, for enter scope senders, is most properties, see above). Ensures that there's only one set_value_t completion which has a single "result datum" which models `exit_scope_sender_in` (i.e. enter scope senders complete successfully by yielding the action which must be taken to exit the scope).

`object<Object>`

Checks environment-independent properties of an async object. Namely that it has a nested type alias called type which indicates the type of object it encapsulates the asynchronous construction of, and that it can be unary invoked with a pointer to said storage yielding an enter scope sender (which constructs an object in that storage and generates an exit scope sender which destroys said object).

`object_in<Object, Env>`

Extends `object` with checks that require an environment. That means checking `enter_scope_sender_in` on the result of unary invocation rather than just `enter_scope_sender` (which is what object checks).

Type Aliases

`exit_scope_sender_of_t<Sender, Env>`

Yields the type of the exit scope sender which is generated by a certain enter scope sender.

`type_of_object_t<Object>`

Determines the type of object constructed by an async object. Equivalent to `std::remove_cvref_t<Object>::type`.

`enter_scope_sender_of_object_t<Object>`

Determines the type of enter scope sender yielded by an async object. Equivalent to `std::invoke_result_t<Object, execution::type_of_object_t<Object>*>`.

Sender Factories

`template<enter_scope_sender... Senders>`
`enter_scope_sender auto enter_scopes(Senders&&... s)`

Produces an enter scope sender which enters the scopes represented by `s...` concurrently.

On the happy path this is essentially (assuming P4217 [23] and P4269 [24] are adopted and LWG4502 is resolved):

```
when_all(s...) | then([](auto&&... s) {
  return when_all(s...);
})
```

On the unhappy path (i.e. when a child completes with error or stopped) any enter scope operations which have completed are rolled back (by running the generated exit scope sender) before that error or stopped completion is sent onwards.

`template<enter_scope_sender... Senders>`
`enter_scope_sender auto nest_scopes(Senders&&... s)`

Produces an enter scope sender which enters the scopes represented by `s...` in sequence.

On the happy path this is essentially (assuming P4320 [16] is adopted):

```
sequence(s...) | then([](auto&&... s) {
  return sequence(REVERSE(s)...);
};
```

Where we imagine *`REVERSE`* reverses a pack.

On the unhappy path (i.e. when a child completes with error or stopped) all entered scopes are exited in reverse order and the unhappy completion is sent onwards.

```
template<sender Sender, enter_scope_sender Scope>
sender auto within(Sender&& sender, Scope&& scope)
```

First enters the scope represented by the provided enter scope sender. Once asynchronous execution reaches the point whereat the enter scope sender's operation has run and an exit scope sender has been produced all pathways to completion involve running that exit scope sender (i.e. users are guaranteed that if the scope is entered it will be exited).

Then connects and starts the asynchronous operation represented by `sender`.

To reiterate: If the asynchronous operation represented by `sender` completes with value, error, or stopped the scope will be asynchronously exited before that completion is sent onwards. If moving `sender` throws the scope will be asynchronously exited before that exception is sent onwards. If connecting `sender` throws the scope will be asynchronously exited before that exception is sent onwards.

Put differently: Runs `sender` **within** the scope represented by `scope`.

Note that this is pipeable:

```
just() |
  then([]() { std::cout << "Holding lock" << std::endl; }) |
  within(gate.acquire())
```

Acquires the gate (see P4215 [22]), prints to the standard output stream, and then releases the gate.

```
template<typename SenderFactory, typename F,
  object... Objects>
sender auto lifetime(SenderFactory&& factory, F&& f,
  Objects&&... objects)
```

Provides the asynchronous analogue of variables with automatic storage duration by asynchronously constructing objects in storage provided by its operation state and allowing those objects to be consumed by some child operation.

Since the storage for the asynchronous objects does not exist until the operation state exists invocation of each of `objects...` must be deferred until at least when `connect` is called (because the operation state is only created by such invocation). This leaves three possibilities:

- Invoke each of `objects...` after the invocation of `start` (i.e. decay-copy each of `objects...` into the operation state so invocation thereof can be deferred)
- Invoke each of `objects...` as part of `connect` (e.g. in the constructor of the operation state) but do not connect the resulting enter scope senders until after the invocation of

`start` (i.e. decay-copy the result of each invocation into the operation state so connection thereof can be deferred)
- As part of `connect` both invoke each of `objects...` and connect the resulting enter scope senders

All three approaches above would achieve the high-level goal of this algorithm. This paper chooses the alternative described in the last bullet because:

- Unlike the first bullet it avoids the cost of:
    - Decay-copying each of `objects...`
    - Storing each of `objects...` in the operation state (note this cost is both in terms of space and implementation complexity)
- Unlike the second bullet it avoids the cost of:
    - Storing each enter scope sender
    - Catching and propagating exceptions resulting from connecting each enter scope sender during the asynchronous part of the operation
- Unlike both the first and second bullets it's arguably more idiomatic since, where possible, algorithms currently in the standard connect their child operations within their own `connect`

The fact that there are multiple asynchronous objects means there are multiple enter scope senders. This raises the question of how these will be combined. The first parameter, `factory`, provides an answer: The enter scope senders resulting from invoking the asynchronous objects will be combined into a single enter scope sender through invocation of `factory`. This provides another argument for the choice made above as factory does not need to be stored in the operation state if invocation of the asynchronous objects, and connection of the resulting enter scope senders, is done within `connect`.

Once the combined enter scope sender's asynchronous operation has been allowed to run the overall lifetime operation is left with:

- An exit scope sender (which destroys all the managed objects), and
- The managed objects

The fact the managed objects are not available until this point means that whatever asynchronous operation will consume them must be late bound, i.e. it must receive references to the managed objects during the asynchronous operation. This is the purpose of `f`. Much like the invocable provided to `let_value` `f` is invoked with each of the managed objects, and returns a sender which represents an asynchronous operation which uses those objects. lifetime then connects that operation, starts it, allows it to run, and only thereafter connects and starts the exit scope senders, allows them to run, and then finally forwards whatever completion was sent by the asynchronous operation resulting from the sender returned by `f`.

```
template<typename F, object... Objects>
sender auto lifetime(F&& f, Objects&&... objects)
```

Equivalent to `lifetime(enter_scopes, f, objects...)`.

Note that in the author's experience this has always been the desired use of `lifetime`. There's a suggested poll on whether to retain the other, more general version.

Classes

`sync_object<T, Args...>`

This is a minimal archetype of the `execution::object` & `::object_in` concepts. It simply allows regular C++ objects with regular synchronous constructors and destructors to be managed by `execution::lifetime`.

This utility provides a superior alternative to the idiom of `execution::just(...) | execution::let_value(...)` ([10] at 40:06) which is commonly used to place one or more objects in an operation state and thereby attach them to an asynchronous scope. Note that this utility is superior to the aforementioned idiom because:

- It does not require that the object(s) be movable (ibid.)
- It has lower storage requirements in the status quo [11]

Put differently, whereas local variables with automatic storage duration are attached to the scope of the regular, synchronous function which contains them, `execution::lifetime(..., execution::sync_object<...>(...))` attaches a variable to the scope of the asynchronous function represented by the sender yielded by the invocable which is the first argument to `execution::lifetime`.

Functions

```
template<typename T, typename... Args>
sync_object<T, decay_t<Args>...> make_sync_object(Args&&...)
```

Factory for `sync_object` since partial CTAD doesn't exist.

Like existing `make_` factory functions (for example `make_optional` (§22.5.10 [optional.specalg])) this function decays the provided arguments. One can leverage `sync_object` to construct objects whose constructors accept references by:

- Passing `std::reference_wrapper` to `make_sync_object`, or

- Instantiating `sync_object` directly with one or more template arguments which are reference types

Examples

`let_async_scope`

As mentioned earlier in the paper the join operation of `execution::simple_counting_scope` and `::counting_scope` is simply an asynchronous destructor. Therefore one of the value adds of the proposed `std::execution::let_async_scope` [5] can be provided via an async object:

```
template<typename Scope>
struct scope_object {
  using type = Scope;

  execution::enter_scope_sender auto operator()(type* storage) const noexcept
  {
    return
      execution::just(storage) |
      execution::then([](type* storage) {
        const auto ptr = new(storage) type();
        return
          ptr->join() |
          execution::then([ptr]() noexcept {
            ptr->~type();
          });
      });
  }

};
```

Asynchronous Mutex

A straightforward design of an asynchronous mutex using `std::execution` might look like this ([10] at 29:37):

```
struct async_mutex {
  template<std::execution::sender Sender>
  std::execution::sender auto with(Sender&& sender);
};
```

Notice that the `with` member function, which is a sender factory, isn't nullary. This is because it needs to wrap another operation so that it can:

- Acquire the lock before running that other operation, and
- Release the lock after running that other operation

Notably the shape of the above API makes it difficult (or impossible) to acquire multiple mutexes in parallel. But also note that the above description of the `with` member function mirrors our understanding of a scope exactly: An action taken upon entering the scope, and a corresponding action taken upon leaving the scope.

With the above understanding, and the primitives proposed by this paper, we can reformulate our asynchronous mutex as:

```
struct async_mutex {
  std::execution::enter_scope_sender auto acquire();
};
```

Now the mutex can be acquired and released, and an operation run while holding the lock, using the `within` operation proposed by this paper:

```
async_mutex m;
auto critical_section =
  std::execution::just() |
  std::execution::then([&]() {
    std::cout << "Holding the async mutex" << std::endl;
  });
auto snd = std::execution::within(m.acquire(), std::move(critical_section));
```

But this only mirrors the functionality available with the previous design. What if we wish to acquire and release several mutexes at the same time? With the new design this is trivial:

```
async_mutex a, b, c;
auto critical_section =
  std::execution::just() |
  std::execution::then([&]() {
    std::cout << "Holding all async mutexes" << std::endl;
  });
auto snd = std::execution::within(
  std::execution::enter_scopes(a.acquire(), b.acquire(), c.acquire()),
  std::move(critical_section));
```

But note that the above assumes a regular, synchronous, exterior scope in which the mutex objects can safely live. That is, there's an implicit assumption that `snd` will be connected, started, and run to completion before `a`, `b`, or `c` go out of scope. As mentioned in the discussion

of the proposed `execution::sync_object`, one way to workaround this in the status quo is with `execution::just(...) | execution::let_value(...)`:

```
auto snd =
  std::execution::just(async_mutex{}, async_mutex{}, async_mutex{}) |
  std::execution::let_value([](
    async_mutex& a, async_mutex& b, async_mutex& c)
  {
    auto critical_section =
      std::execution::just() |
      std::execution::then([&]() {
        std::cout << "Holding all async mutexes" << std::endl;
      });
    return std::execution::within(
      std::execution::enter_scopes(a.acquire(), b.acquire(), c.acquire()),
      std::move(critical_section));
  });
```

This places the three asynchronous mutexes in the operation state formed when the sender is connected, thereby ensuring that the mutexes remain valid for the lifetime of the resulting asynchronous operation. Unfortunately however it assumes that `async_mutex` is movable to:

- Store the mutexes in `execution::just`,
- Compose `execution::just` with `execution::let_value`,
- Connect `snd`, and
- Propagate the mutexes from the operation state of `execution::just` to the operation state of `execution::let_value`

Not only is that quite a few moves, but mutexes are generally not movable. In the status quo this would be difficult to workaround but `execution::sync_object` provides a solution:

```
auto snd = std::execution::lifetime(
  [](async_mutex& a, async_mutex& b, async_mutex& c) {
    auto critical_section =
      std::execution::just() |
      std::execution::then([&]() {
        std::cout << "Holding all async mutexes" << std::endl;
      });
    return std::execution::within(
      std::execution::enter_scopes(a.acquire(), b.acquire(), c.acquire()),
      std::move(critical_section));
  },
  std::execution::sync_object<async_mutex>{},
```

```
  std::execution::sync_object<async_mutex>{},
  std::execution::sync_object<async_mutex>{});
```

This works because `execution::sync_object<async_mutex>{}`, unlike `async_mutex{}`, doesn't actually construct an `async_mutex`. Instead it curries arguments to an asynchronous constructor which doesn't run until the sender yielded by `execution::lifetime` is connected and the resulting operation state is started. Since `execution::lifetime` asynchronously constructs and destroys async objects in storage within its operation state these `async_mutex` objects are never moved and therefore the fact that they are immovable is not an issue.

Note: This example has been present in this paper since R0. Since the publication of R0 of this paper P4215 [21] has proposed an entire set of such primitives built on top of the design of this paper. The `async_mutex` above corresponds to `serial_gate` from P4215.

io_uring Socket Pair

The below example uses io_uring [12] to:

1. Asynchronously create two sockets (via `IORING_OP_SOCKET`),
2. Listen on one of the sockets (via `IORING_OP_BIND` and `IORING_OP_LISTEN`),
3. Connect to the socket from step 2 from the other socket (via `IORING_OP_CONNECT` on the connecting side and `IORING_OP_ACCEPT` on the listening side),
4. Send a string from the socket connected in step 3 (via `IORING_OP_WRITE`),
5. Read that string from the socket accepted in step 3 (via `IORING_OP_READ`), and then
6. Asynchronously destroy all three sockets (listening, connecting, and accepted) (via `IORING_OP_CLOSE`)

```
const std::string_view sv("Hello world!");
std::vector<std::byte> buffer(sv.size());
std::this_thread::sync_wait(
  run_on_blocking_io_uring(
    [&](io_uring_context& ctx) {
      const socket_object object(
        ctx,
        AF_INET,
        SOCK_STREAM,
        IPPROTO_TCP);
      return std::execution::lifetime(
        [&](file_descriptor& server, file_descriptor& client)
          -> std::execution::task<void>
        {
          sockaddr_in addr;
          std::memset(&addr, 0, sizeof(addr));
          addr.sin_family = AF_INET;
```

```
        co_await bind(server, addr);
        co_await listen(server, SOMAXCONN);
        socklen_t out = sizeof(addr);
        if (getsockname(
          server.native_handle(),
          reinterpret_cast<sockaddr*>(&addr),
          &out) == -1)
        {
          throw std::runtime_error("getsockname failed");
        }
        co_await std::execution::when_all(
          [&]() -> std::execution::task<void> {
            co_await connect(client, addr);
            co_await write(
              client,
              std::span{
                reinterpret_cast<const std::byte*>(sv.data()),
                sv.size()});
          }(),
          std::execution::lifetime(
            [&](file_descriptor& accepted) {
              return read(accepted, buffer);
            },
            accept_object(server)));
      },
      object,
      object);
  },
  32,
  io_uring_params{}));
```

Note that io_uring operations which simply perform I/O (`IORING_OP_CONNECT`, `IORING_OP_WRITE`, et cetera) are presented such that they yield a regular sender (which can be `co_awaited` due to `std::execution::task` [13]) whereas io_uring operations which create a file descriptor (`IORING_OP_SOCKET` and `IORING_OP_ACCEPT`) are wrapped by an async object and reified via `execution::lifetime` to ensure the proper asynchronous destruction occurs on scope exit.

Importantly the above code is materially different from (note important differences in **bold**):

```
const std::string_view sv("Hello world!");
std::vector<std::byte> buffer(sv.size());
std::this_thread::sync_wait(
  run_on_blocking_io_uring(
```

```
[&](io_uring_context& ctx) {
  file_descriptor server(
    ctx,
    co_await socket(
      ctx,
      AF_INET,
      SOCK_STREAM,
      IPPROTO_TCP));
  file_descriptor client(
    ctx,
    co_await socket(
      ctx,
      AF_INET,
      SOCK_STREAM,
      IPPROTO_TCP));
  sockaddr_in addr;
  std::memset(&addr, 0, sizeof(addr));
  addr.sin_family = AF_INET;
  co_await bind(server, addr);
  co_await listen(server, SOMAXCONN);
  socklen_t out = sizeof(addr);
  if (getsockname(
    server.native_handle(),
    reinterpret_cast<sockaddr*>(&addr),
    &out) == -1)
  {
    throw std::runtime_error("getsockname failed");
  }
  co_await std::execution::when_all(
    [&]() -> std::execution::task<void> {
      co_await connect(client, addr);
      co_await write(
        client,
        std::span{
          reinterpret_cast<const std::byte*>(sv.data()),
          sv.size()});
    }(),
    [&]() -> std::exceution::task<void> {
      file_descriptor accepted(
        ctx,
        co_await accept(server));
      co_await read(accepted, buffer);
    }());
```

```
    },
    32,
    io_uring_params{}));
```

Because in this example `file_descriptor` has been reimagined as a regular, synchronous object with a regular, synchronous destructor. Therefore the synchronous close syscall must be used for clean up rather than the asynchronous `IORING_OP_CLOSE`. Note that in both cases the sockets are constructed asynchronously (via `IORING_OP_SOCKET` and `IORING_OP_ACCEPT`), it is only in destruction that the examples materially differ.

Proposed Wording

[execution.syn]

```
  // [exec.parschedrepl], namespace parallel_scheduler_replacement
  namespace parallel_scheduler_replacement {
    struct receiver_proxy;
    struct bulk_item_receiver_proxy;
    struct parallel_scheduler_backend;

    shared_ptr<parallel_scheduler_backend>
      query_parallel_scheduler_backend();
  }

  // [exec.scopesndrs.concepts] Scope sender concepts
  template<class Sndr>
  concept exit_scope_sender = see below;
  template<class Sndr, class Env>
  concept exit_scope_sender_in = see below;

  template<class Sndr>
  concept enter_scope_sender = see below;

  template<enter_scope_sender Sndr, class Env>
  using exit_scope_sender_of_t = see below;

  template<class Sndr, class Env>
  concept enter_scope_sender_in = see below;

  // [exec.scopesndrs.algos.within]
  struct within_t { unspecified };
  inline constexpr within_t within{};
```

```
  // [exec.scopesndrs.algos.compose]
  struct enter_scopes_t { unspecified };
  struct nest_scopes_t { unspecified };
  inline constexpr enter_scopes_t enter_scopes{};
  inline constexpr nest_scoped_t nest_scopes{};

  // [exec.objects.concepts] Async object concepts
  template<class Object>
  concept object = see below;
  template<class Object, class Env>
  concept object_in = see below;

  template<class Object>
  using type_of_object_t = see below;

  template<class Object>
  using enter_scope_sender_of_object_t = see below;

  // [exec.objects.algos.lifetime]
  struct lifetime_t { unspecified };
  inline constexpr lifetime_t lifetime{};

  // [exec.object.sync] Asynchronous adapter for synchronous objects
  template<class Object, class... Ts>
    requires constructible_from<Object, Ts...>
  struct sync_object;

  template<class Object, class... Ts>
  sync_object<Object, decay_t<Ts>...> make_sync_object(Ts&&... ts) noexcept(
    (is_nothrow_constructible_v<decay_t<Ts>, Ts> && ...));
}
```

Note to the Editor

All wording which follows is additive.

[exec.scopesndrs]

An *enter scope sender* is a higher-order sender whose asynchronous operation establishes some invariant upon successful completion. Successful completion of the asynchronous operation associated with an enter scope sender produces an *exit scope sender* which

represents the asynchronous work required to undo the invariant established by the enter scope sender.

Among other things, enter and exit scope senders may be used to model asynchronous resource lifetimes.

The utilities in this section provide vocabulary for identifying enter and exit scope senders, for composing them, and for using them safely.

[exec.scopesndrs.concepts]

Note: The below wording assumes the adoption of P4191 [14].

```
template<class Sndr>
concept exit_scope_sender =
  sender<Sndr>                                          &&
  is_nothrow_constructible_v<remove_cvref_t<Sndr>, Sndr> &&
  is_nothrow_move_constructible_v<remove_cvref_t<Sndr>>;

template<class Sndr, class Env>
concept exit_scope_sender_in =
  exit_scope_sender<Sndr>                &&
  sender_in<Sndr, Env>                   &&
  is_nothrow_connectable_in_v<Sndr, Env> &&
  same_as<
    completion_signatures_of_t<Sndr, Env>,
    completion_signatures<set_value_t()>>;

template<class Sndr>
concept enter_scope_sender = sender<Sndr>;
```

For types `Sndr` and `Env`, if `sender_in<Sndr, Env>` is `false`, the alias `exit_scope_sender_of_t<Sndr, Env>` does not denote a type. Otherwise, let `Sigs` be `completion_signatures_of_t<Sndr, Env>`, and let `Vs` be the pack of types in `Sigs` whose completion function is `set_value_t`. If `sizeof...(Vs)` is not `1` then `exit_scope_sender_of_t<Sndr, Env>` does not denote a type. Otherwise, let `Ts` be the pack such that `set_value_t(Ts...)` is the sole type in `Vs`. If `sizeof...(Ts)` is not `1` then `exit_scope_sender_of_t<Sndr, Env>` does not denote a type. Otherwise let `Exit` be the sole type in `Ts`. If `exit_scope_sender_in<Exit, Env>` is `false`, then `exit_scope_sender_of_t<Sndr, Env>` does not denote a type. Otherwise, `exit_scope_sender_of_t<Sndr, Env>` denotes `Exit`.

```
template<class Sndr, class Env>
concept enter_scope_sender_in =
```

```
  enter_scope_sender<Sndr> &&
  sender_in<Sndr, Env>     &&
  requires {
    typename exit_scope_sender_of_t<Sndr, Env>;
  };
```

[exec.scopesndrs.algos.within]

Note: The below wording assumes the adoption of P4288 [15].

`within` enters an asynchronous scope represented by an enter scope sender ([exec.scopesndrs]), starts an asynchronous operation while within that scope, and then exits the scope before completing with the completion of the child operation.

The name `within` denotes a pipeable sender adaptor object. For subexpressions `sndr` and `scope`, if `decltype((sndr))` does not satisfy `sender` or `decltype((scope))` does not satisfy `scope`, the expression `within(sndr, scope)` is ill-formed. Otherwise, the expression `within(sndr, scope)` is expression-equivalent to:

`make-sender``(within, {}, sndr, scope)`

Let *`within-tag`* denote a unique, empty class type. Let *`within-data`* denote the following exposition-only class template:

```
template<class Body, class Scope>
struct within-data {
  Body body;
  Scope scope;
};
```

The expression `within.transform_sender(s, env)` is expression-equivalent to:

```
make-sender(
  within-tag{},
  within-data{s.template get<1>(), s.template get<2>()})
```

except that `s` is evaluated only once.

The exposition-only class template *`impls-for`* ([exec.snd.expos]) is specialized for *`within-tag`* as follows:

```
template<>
struct impls-for<within-tag> : default-impls {
  static constexpr auto get-state = see below;
  static constexpr auto start = see below;
```

```
  template<class Sndr, class... Env>
    static consteval void check-types();
};
```

Let *within-state* denote the following exposition-only class template:

```
template<class Body, class Env>
using within-storage-t = see below;                                  // exposition only

template<class Body, class Scope, class Rcvr>
struct within-state {
  using env-t = env_of_t<Rcvr>;                                      // exposition only
  using exit-t = exit_scope_sender_of_t<Scope, env-t>;               // exposition only
  using storage-t = within-storage-t<Body, env-t>;                   // exposition only

  struct enter-receiver {                                            // exposition only
    using receiver_concept = receiver_tag;

    within-state& state;                                             // exposition only
    Rcvr& rcvr;                                                      // exposition only

    constexpr void set_value(exit-t exit) && noexcept {
      try {
        auto body = std::move(get<enter-state>(state.op).body);
        auto mkstate = [&] {
          return body-state{
            connect(std::move(body), body-receiver{state, rcvr}),
            std::move(exit)};
        };
        start(state.op.template emplace<body-state>(emplace-from{mkstate}));
      } catch (...) {
        if constexpr (!is_nothrow_connectible_to_v<
          remove_cvref_t<Body>, body-receiver>)
        {
          state.storage.arrive(set_error, current_exception());
          state.exit(rcvr, std::move(exit));
        }
      }
    }

    template<class E>
    constexpr void set_error(E&& e) && noexcept {
      execution::set_error(std::move(rcvr), std::forward<E>(e));
```

```
  }

  constexpr void set_stopped() && noexcept {
    execution::set_stopped(std::move(rcvr));
  }

  constexpr env-t get_env() const noexcept {
    return execution::get_env(rcvr);
  }
};

struct body-receiver {                                          // exposition only
  using receiver_concept = receiver_tag;

  within-state& state;                                          // exposition only
  Rcvr& rcvr;                                                   // exposition only

  template<class... Args>
  constexpr void set_value(Args&&... args) && noexcept {
    state.body-complete(rcvr, set_value, std::forward<Args>(args)...);
  }

  template<class E>
  constexpr void set_error(E&& e) && noexcept {
    state.body-complete(rcvr, set_error, std::forward<E>(e));
  }

  constexpr void set_stopped() && noexcept {
    state.body-complete(rcvr, set_stopped);
  }

  constexpr env-t get_env() const noexcept {
    return execution::get_env(rcvr);
  }
};

struct exit-receiver {                                          // exposition only
  using receiver_concept = receiver_tag;

  within-state& state;                                          // exposition only
  Rcvr& rcvr;                                                   // exposition only

  constexpr void set_value() && noexcept {
```

```
    state.storage.complete(std::move(rcvr));
  }

  constexpr env-t get_env() const noexcept {
    return execution::get_env(rcvr);
  }
};

using enter-op-t = connect_result_t<Scope, enter-receiver>;  // exposition only
using body-op-t =
  connect_result_t<remove_cvref_t<Body>, body-receiver>;     // exposition only
using exit-op-t = connect_result_t<exit-t, exit-receiver>;   // exposition only

struct enter-state {                                         // exposition only
  enter-op-t op;
  remove_cvref_t<Body> body;
};

struct body-state {                                          // exposition only
  body-op-t op;
  exit-t exit;
};

storage-t storage;                                           // exposition only
variant<enter-state, body-state, exit-op-t> op;              // exposition only

constexpr void exit(Rcvr& rcvr, exit-t exit) noexcept {      // exposition only
  auto mkop = [&]() noexcept {
    return connect(std::move(exit), exit-receiver{*this, rcvr});
  };
  start(op.template emplace<exit-op-t>(emplace-from{mkop}));
}

template<class Tag, class... Args>
  constexpr void body-complete(Rcvr& rcvr, Tag tag, Args&&... args)
    noexcept
  {                                                          // exposition only
    storage.arrive(tag, std::forward<Args>(args)...);
    exit(rcvr, std::move(get<body-state>(op).exit));
  }

constexpr within-state(Body&& body, Scope&& scope, Rcvr& rcvr) noexcept(
  is_nothrow_constructible_v<remove_cvref_t<Body>, Body> &&
```

```
    is_nothrow_connectible_to_v<Scope, enter-receiver>)
    : op(in_place_type<enter-state>, emplace-from{[&] {
        return enter-state{
          connect(std::forward<Scope>(scope), enter-receiver{*this, rcvr}),
          std::forward<Body>(body)};
      }}) {}
};
```

For types `Body` and `Env`, if `sender_in<remove_cvref_t<Body>, Env>` is `false`, *`within-storage-t`*`<Body, Env>` does not denote a type. Otherwise, let `BodySigs` be `completion_signatures_of_t<remove_cvref_t<Body>, Env>`. Let `ExtraSigs` denote `completion_signatures<>` if:

- `is_nothrow_move_constructible_v<remove_cvref_t<Body>>` is `true`, and
- `is_nothrow_connectable_in_v<remove_cvref_t<Body>, Env>` is `true`

Otherwise, let `ExtraSigs` denote `completion_signatures<set_error_t(exception_ptr)>`. Let `StorageSigs` denote a specialization of `completion_signatures` whose template arguments are all template arguments of `BodySigs` and `ExtraSigs` except with duplicates removed. *`within-storage-t`*`<Body, Env>` denotes `storage_for_completion_signatures<StorageSigs>`.

`impls-for``<`*`within-tag`*`>::`*`get-state`* is initialized with a callable object equivalent to the following:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept(see below) {
  auto& [_, data] = sndr;
  auto&& [body, scope] = std::forward_like<Sndr>(data);
  return within-state<decltype(body), decltype(scope), Rcvr>(
    std::forward_like<Sndr>(body),
    std::forward_like<Sndr>(scope),
    rcvr);
}
```

Let *e* denote the expression that is the operand of the `return` statement. The expression in the `noexcept` clause is equivalent to `noexcept(`*`e`*`)`.

`impls-for``<`*`within-tag`*`>::`*`start`* is initialized with a callable object equivalent to the following:

```
[]<class State, class Rcvr>(State& state, Rcvr&) noexcept {
  start(get<typename State::enter-state>(state.op).op);
}
```

```
template<class Sndr, class... Env>
  static consteval void check-types();
```

Effects: Equivalent to:

```
if constexpr (sizeof...(Env) == 0) {
  throw dependent_sender_error();
} else {
  using body_t =
    remove_cvref_t<decltype(declval<Sndr>().template get<1>().body)>;
  using scope_t = decltype(declval<Sndr>().template get<2>().scope);
  if constexpr (
    enter_scope_sender_in<scope_t, Env...> &&
    sender_in<body_t, Env...>)
  {
    within-storage-t<body_t, Env...>::get_completion_signatures();
  } else {
    throw unspecified-exception();
  }
}
```

[exec.scopesndrs.algos.compose]

Note: The below wording assumes the adoption of P4320 [16] and P4329 [17].

`enter_scopes` and `nest_scopes` compose enter scope senders ([exec.scopesndrs]) into a single enter scope sender. `enter_scopes` enters the input scopes concurrently. `nest_scopes` enters the input scopes sequentially. The exit scope sender produced by either algorithm exits the input scopes; in the case of `enter_scopes` the input scopes are exited concurrently, in the case of `nest_scopes` the input scopes are exited sequentially in the reverse of the order in which they were entered.

Let *`scope-cpo`* be one of `enter_scopes` or `nest_scopes`. For `enter_scopes` and `nest_scopes`, let *`scope-algo`* be `when_all` and `sequence`, respectively.

The names `enter_scopes` and `nest_scopes` denote sender adaptor objects. For a pack of subexpressions `sndrs...`, *`scope-cpo`*`(sndrs...)` is ill-formed if `(enter_scope_sender<decltype((sndrs))> && ...)` is not `true`. Otherwise, *`scope-cpo`*`(sndrs...)` is expression-equivalent to:

- `just(just())` if `sizeof...(sndrs)` is `0`,
- `sndr` if `sizeof...(sndrs)` is `1`, where `sndr` is the sole subexpression in `sndrs`,
- *`make-sender`*`(`*`scope-cpo`*`, {}, sndrs...)` otherwise.

Let *`scope-tag`* denote a unique, empty class type for each of `enter_scopes` and `nest_scopes`. The expression *`scope-cpo`*`.transform_sender(s, env)` is expression-equivalent to:

make-sender(*scope-tag*{}, tuple(sndrs...))

where sndrs is the pack that would be introduced by:

```
auto&& [_, _, ...sndrs] = s;
```

Let *variant-sender-for* denote the following exposition-only alias template:

```
template<class Sndr>
  using variant-sender-for =
    variant_sender<decltype(just()), remove_cvref_t<Sndr>>;
```

Let *make-variant-sender* denote the following exposition-only function template:

```
template<class Sndr>
  constexpr decltype(auto) make-variant-sender(optional<Sndr>&& sndr)
    noexcept
  {
    if (sndr) {
      return variant-sender-for<Sndr>(std::move(*sndr));
    }
    return variant-sender-for<Sndr>(just());
  }
```

Let *reverse-scope-algo*(sndrs...) be expression-equivalent to:

scope-algo(rsndrs...)

where rsndrs... is expression-equivalent to sndrs... except with the elements of the pack of expressions in the reverse order.

For a type Env and a pack of types Sndrs, let *storage-t*<Env, Sndrs...> denote a type as follows. Let EnterSigs be completion_signatures_of_t<decltype(*scope-algo*(declval<Sndrs>()...)), Env>. Let StorageSigs denote a specialization of completion_signatures whose template arguments are the template arguments of EnterSigs whose completion function is not set_value_t. The type *storage-t*<Env, Sndrs...> is storage_for_completion_signatures<StorageSigs>.

Let *scope-compose-state* denote the following exposition-only class template:

```
template<class, class, class...>
struct scope-compose-state;

template<class Rcvr, size_t... Is, class... Sndrs>
struct scope-compose-state<Rcvr, index_sequence<Is...>, Sndrs...> {
```

```
using env-t = env_of_t<Rcvr>;                                   // exposition only
using exit-senders-t = tuple<
  optional<exit_scope_sender_of_t<Sndrs, env-t>>...>;           // exposition only

template<size_t I>
struct store-exit-sender {                                      // exposition only
  scope-compose-state& state;                                   // exposition only

  template<class Sndr>
  constexpr void operator()(Sndr&& sndr) noexcept {
    std::get<I>(state.exit-senders) = std::forward<Sndr>(sndr);
  }
};

struct enter-receiver {                                         // exposition only
  using receiver_concept = receiver_tag;

  scope-compose-state& state;                                   // exposition only
  Rcvr& rcvr;                                                   // exposition only

  constexpr void set_value() && noexcept {
    execution::set_value(
      std::move(rcvr),
      reverse-scope-algo(
        *std::move(std::get<Is>(state.exit-senders))...));
  }

  template<class E>
  constexpr void set_error(E&& e) && noexcept {
    state.rollback(rcvr, execution::set_error, std::forward<E>(e));
  }

  constexpr void set_stopped() && noexcept {
    state.rollback(rcvr, execution::set_stopped);
  }

  constexpr env-t get_env() const noexcept {
    return execution::get_env(rcvr);
  }
};

struct exit-receiver {                                          // exposition only
  using receiver_concept = receiver_tag;
```

```
  scope-compose-state& state;                                  // exposition only
  Rcvr& rcvr;                                                  // exposition only

  constexpr void set_value() && noexcept {
    std::move(state.storage).complete(std::move(rcvr));
  }

  constexpr env-t get_env() const noexcept {
    return execution::get_env(rcvr);
  }
};

using enter-t = decltype(
  scope-algo(
    declval<Sndrs>() | then(
      declval<store-exit-sender<Is>>())...));                  // exposition only

using enter-op-t =
  connect_result_t<enter-t, enter-receiver>;                   // exposition only

using exit-t = decltype(
  reverse-scope-algo(
    declval<
      variant-sender-for<
        exit_scope_sender_of_t<Sndrs, env-t>>>()...));         // exposition only

using exit-op-t =
  connect_result_t<exit-t, exit-receiver>;                     // exposition only

template<class... Args>
constexpr void rollback(Rcvr& rcvr, Args&&... args) noexcept
{                                                              // exposition only
  storage.arrive(std::forward<Args>(args)...);
  auto mkop = [&] noexcept {
    return connect(
      reverse-scope-algo(
        make-variant-sender(
          std::move(std::get<Is>(exit-senders)))...),
      exit-receiver{*this, rcvr});
  };
  start(op.template emplace<exit-op-t>(emplace-from{mkop}));
}
```

```
  exit-senders-t exit-senders;                                  // exposition only
  storage-t<env-t, Sndrs...> storage;                           // exposition only
  variant<enter-op-t, exit-op-t> op;                            // exposition only

  template<class... Ts>
  constexpr scope-compose-state(Ts&&... ts, Rcvr& rcvr) noexcept(
    is_nothrow_connectible_to_v<enter-t, enter-receiver> &&
    (is_nothrow_constructible_v<Sndrs, Ts> && ...))
    : op(in_place_type<enter-op-t>, emplace-from{[&] {
        return connect(
          scope-algo(
            std::forward<Ts>(ts) |
            then(store-exit-sender<Is>{*this})...),
          enter-receiver{*this, rcvr});
      }}) {}
};
```

impls-for<*scope-tag*>::*get-state* is initialized with a callable object equivalent to the following:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept(see below) {
  auto& [_, data] = sndr;
  auto&& [...sndrs] = std::forward_like<Sndr>(data);
  return scope-compose-state<
    Rcvr,
    make_index_sequence<sizeof...(sndrs)>,
    remove_cvref_t<decltype(sndrs)>...>(
      std::forward<decltype(sndrs)>(sndrs)...,
      rcvr);
}
```

Let *e* denote the expression that is the operand of the `return` statement. The expression in the noexcept clause is equivalent to `noexcept(`*e*`)`.

`impls-for<scope-tag>::start` is initialized with a callable object equivalent to the following:

```
[]<class State, class Rcvr>(State& state, Rcvr&) noexcept {
  start(get<typename State::enter-op-t>(state.op));
}
```

```
template<class Sndr, class... Env>
  static consteval void check-types();
```

Effects: Let `sndr` denote an lvalue of type `Sndr`. Let `Sndrs` denote the pack `decltype(sndrs)...`, where `sndrs` is the pack introduced by:

```
auto& [_, tuple] = sndr;
auto&& [...sndrs] = std::forward_like<Sndr>(tuple);
```

Equivalent to:

```
if constexpr (sizeof...(Env) == 0) {
  throw dependent_sender_error();
} else {
  if constexpr (
    ((enter_scope_sender_in<remove_cvref_t<Sndrs>, Env...> &&
      is_constructible_v<remove_cvref_t<Sndrs>, Sndrs>) && ...))
  {
    storage-t<Env..., remove_cvref_t<Sndrs>...>::get_completion_signatures();
  } else {
    throw unspecified-exception();
  }
}
```

[exec.objects]

An *asynchronous object* is a unary enter scope sender factory. The sole argument to the factory is a pointer to storage wherein the asynchronous object is to be constructed. The returned enter scope sender ([exec.scopesndrs]) represents an asynchronous operation which constructs an object in that storage, and yields an exit scope sender which represents an asynchronous operation which destroys that object.

[exec.objects.concepts]

```
template<class Object>
concept object =
  constructible_from<remove_cvref_t<Object>, Object> &&
  move_constructible<remove_cvref_t<Object>>         &&
  is_object_v<typename remove_cvref_t<Object>::type> &&
  requires(Object&& object) {
    {
      std::forward<Object>(object)(
        static_cast<typename remove_cvref_t<Object>::type*>(nullptr))
    } -> enter_scope_sender;
  };
```

```
template<class Object, class Env>
concept object_in =
  object<Object> &&
  requires(Object&& object) {
    {
      std::forward<Object>(object)(
        static_cast<type_of_object_t<Object>*>(nullptr))
    } -> enter_scope_sender_in<Env>;
  };

template<object Object>
using type_of_object_t = typename remove_cvref_t<Object>::type;

template<object Object>
using enter_scope_sender_of_object_t =
  invoke_result_t<Object, type_of_object_t<Object>*>;
```

[exec.objects.algos.lifetime]

`lifetime` creates a sender that asynchronously constructs asynchronous objects, synthesizes an asynchronous operation which may depend on said asynchronous objects, runs that asynchronous operation, and then asynchronously destroys the managed asynchronous objects.

Let *`lifetime-body`* denote the following exposition-only concept:

```
template<class F, class... Objects>
  concept lifetime-body = sender<
    invoke_result_t<decay_t<F>, type_of_object_t<Objects>&...>>;
```

Let *`lifetime-sender-factory`* denote the following exposition-only concept:

```
template<class Factory, class... Objects>
  concept lifetime-sender-factory = enter_scope_sender<
    invoke_result_t<
      decay_t<Factory>,
      enter_scope_sender_of_object_t<remove_cvref_t<Objects>>...>>;
```

Let *`lifetime-data`* denote the following exposition-only class template:

```
template<class Factory, class F, class... Objects>
  struct lifetime-data {
    Factory factory;            // exposition only
    F f;                        // exposition only
```

```
    tuple<Objects...> objects;  // exposition only
  };
```

The name `lifetime` denotes a sender adaptor object. For subexpressions `factory`, `f`, and `objects...`, the expression `lifetime(factory, f, objects...)` is expression-equivalent to:

```
make-sender(
  lifetime,
  lifetime-data<
    decay_t<decltype((factory))>,
    decay_t<decltype((f))>,
    remove_cvref_t<decltype((objects))>...>{
      factory,
      f,
      {objects...}})
```

if *`lifetime-sender-factory`*`<decltype((factory)), decltype((objects))...>` is `true` and *`lifetime-body`*`<decltype((f)), decltype((objects))...>` is `true`. Otherwise for subexpressions `f` and `objects...`, the expression `lifetime(f, objects...)` is expression-equivalent to:

```
lifetime(enter_scopes, f, objects...)
```

if *`lifetime-body`*`<decltype((f)), decltype((objects))...>` is `true`. Otherwise `lifetime(f, objects...)` is ill-formed.

Let *`lifetime-state`* denote the following exposition-only class template:

```
template<typename Object>
struct storage-for {                                  // exposition only
  using type = type_of_object_t<Object>;

  alignas(type) char buffer[sizeof(type)];

  constexpr type* ptr() noexcept {
    return reinterpret_cast<type*>(buffer);
  }

  constexpr type& get() noexcept {
    return *launder(ptr());
  }
};

template<typename F, typename... Objects>
```

```
struct body-wrapper {                                   // exposition only
  F& f;                                                 // exposition only
  tuple<storage-for<Objects>...>& storage;              // exposition only

  constexpr invoke_result_t<F, type_of_object_t<Objects>&...>
    operator()() && noexcept(
      is_nothrow_invocable_v<F, type_of_object_t<Objects>&...>)
  {                                                     // exposition only

    auto& [...slots] = storage;
    return invoke(std::move(f), slots.get()...);
  }
};

template<class F, class... Objects>
using body-sender-t = decltype(                         // exposition only
  just() | let_value(declval<body-wrapper<F, Objects...>>()));

template<class Factory, class... Objects>
using enter-scope-sender-t = invoke_result_t<
  Factory,
  enter_scope_sender_of_object_t<Objects>...>;          // exposition only

template<class Factory, class F, class... Objects>
using within-t = decltype(                              // exposition only
  within(
    declval<enter-scope-sender-t<Factory, Objects...>>(),
    declval<body-sender-t<F, Objects...>>()));

template<class Rcvr, class Factory, class F, class... Objects>
struct lifetime-state {
  F f;                                                  // exposition only
  tuple<storage-for<Objects>...> storage;               // exposition only
  connect_result_t<
    within-t<Factory, F, Objects...>,
    Rcvr> op;                                           // exposition only

  constexpr lifetime-state(
    Rcvr rcvr,
    Factory&& factory,
    F f,
    Objects&&... objects) noexcept(
      is_nothrow_move_constructible_v<F>    &&
```

```
      (is_nothrow_invocable_v<
        Objects,
        type_of_object_t<Objects>*> && ...) &&
      is_nothrow_invocable_v<
        Factory,
        invoke_result_t<
          Objects,
          type_of_object_t<Objects>*>...>   &&
      is_nothrow_invocable_v<
        within_t,
        enter-scope-sender-t<Factory, Objects...>,
        body-sender-t<F, Objects...>>       &&
      is_nothrow_connectible_to_v<within-t<Factory, F, Objects...>, Rcvr>)
    : f(std::move(f)),
      op([&] {
        auto& [...storage] = this->storage;
        return connect(
          within(
            invoke(
              std::forward<Factory>(factory),
              std::forward<Objects>(objects)(storage.ptr())...),
            just() | let_value(body-wrapper<F, Objects...>{
              this->f,
              this->storage})),
          std::move(rcvr));
      }()) {}
};
```

The exposition-only class template *impls-for* ([exec.snd.expos]) is specialized for `lifetime_t` as follows:

```
template<>
struct impls-for<lifetime_t> : default-impls {
  static constexpr auto get-state = see below;
  static constexpr auto start = see below;

  template<class Sndr, class... Env>
    static consteval void check-types();
};
```

impls-for`<lifetime_t>::`*get-state* is initialized with a callable object equivalent to the following:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept(see below) {
    auto& [_, data] = sndr;
    auto&& [factory, f, tuple] = std::forward_like<Sndr>(data);
    auto&& [...objects] = std::forward_like<Sndr>(tuple);
    return lifetime-state<
      Rcvr,
      decltype(factory),
      remove_cvref_t<decltype(f)>,
      decltype(objects)...>(
        std::move(rcvr),
        std::forward_like<Sndr>(factory),
        std::forward_like<Sndr>(f),
        std::forward_like<Sndr>(objects)...);
}
```

Let *e* denote the expression that is the operand of the `return` statement. The expression in the `noexcept` clause is equivalent to `noexcept(`*e*`)`.

impls-for`<lifetime_t>::`*start* is initialized with a callable object equivalent to the following:

```
[]<class State, class Rcvr>(State& state, Rcvr&) noexcept {
  start(state.op);
}
```

```
template<class Sndr, class... Env>
  static consteval void check-types();
```

Effects: Let `sndr` denote an lvalue of type `Sndr`. Let `factory`, `f`, and `objects...` be the bindings introduced by:

```
auto& [_, data] = sndr;
auto& [factory, f, tuple] = std::forward_like<Sndr>(data);
auto&& [...objects] = std::forward_like<Sndr>(tuple);
```

Equivalent to:

```
if constexpr (sizeof...(Env) == 0) {
  throw dependent_sender_error();
} else {
  using factory_t = decltype(factory);
  using f_t = remove_cvref_t<decltype(f)>;
  if constexpr (
    lifetime-sender-factory<factory_t, decltype(objects)...> &&
    lifetime-body<f_t, decltype(objects)...>)
  {
```

```
    get_completion_signatures<
      within-t<factory_t, f_t, decltype(objects)...>,
      Env...>();
  } else {
    throw unspecified-exception();
  }
}
```

[exec.object.sync]

Specializations of the `sync_object` class template combine the type of an object with arguments to one of that object's constructors thereby allowing that object to participate in asynchronous construction and destruction ([exec.objects.algos.lifetime]).

```
template<class Object, class... Ts>
  requires constructible_from<Object, Ts...>
struct sync_object {
  tuple<Ts...> ts;  // exposition only

  template<typename... Us>
    requires (constructible_from<Ts, Us> && ...)
  explicit constexpr sync_object(Us&&... us) noexcept(
    (is_nothrow_constructible_v<Ts, Us> && ...)))
    : ts(std::forward<Us>(us)) {}

  template<class Self>
    requires constructible_from<
      tuple<Ts...>,
      decltype(std::forward_like<Self>(declval<tuple<Ts...>>()))>
  constexpr enter_scope_sender auto operator()(
    this Self&& self,
    Object* storage) noexcept(
      is_nothrow_constructible_v<
        tuple<Ts...>,
        decltype(std::forward_like<Self>(declval<tuple<Ts...>>()))>)
  {
    return just() | then([storage, ts = std::forward_like<Self>(ts)]()
      mutable noexcept(is_nothrow_constructible_v<Object, Ts...>) ->
        exit_scope_sender auto
    {
      auto&& [...args] = std::move(ts);
      const auto ptr = construct_at(
        storage,
```

```
        std::forward<decltype(args)>(args));
      return just() | then([ptr]() noexcept {
        destroy_at(ptr);
      });
    });
  }
};

template<class Object, class... Ts>
sync_object<Object, decay_t<Ts>...> make_sync_object(Ts&&... ts) noexcept(
  (is_nothrow_constructible_v<decay_t<Ts>, Ts> && ...))
{
  return sync_object<Object, decay_t<Ts>...>(std::forward<Ts>(ts)...);
}
```

Implementation Experience

The author has implemented this design on top of nVidia's stdexec [18].

Additional Material

The author has detailed this design in a C++Now talk [19].

Suggested/Desired Polls

Construction Order Customization

`lifetime` should not allow customization of the means by which enter scope senders are combined

This would specify the interface and the specification. There is no known use case for any enter scope combination algorithm other than `enter_scopes` (i.e. the author has never encountered a situation wherein the construction of the various async objects was relevant).

Binary Concepts

`exit_scope_sender_in` should mandate (i.e. hard error rather than concept evaluating to `false`) that the sender is nothrow connectable (this would allow `object_in`,

`enter_scope_sender_in`, and `exit_scope_sender_in` to become variadic consistently with `sender_in`)

This would permit earlier (i.e. non-dependent) diagnostics [20]. Note that one can conceive of a universe wherein senders can advertise whether or not they can bo nothrow connected without the context of a receiver or environment but that is not the design which was chosen [8].

Mandate `noexcept`

`exit_scope_sender_in` should mandate (i.e. hard error rather than concept evaluating to `false`) that the sender is nothrow decay copyable and move constructible

Eric Niebler has suggested that users are likely to forget `noexcept`, thereby causing the concept to evaluate to `false`, thereby causing opaque and difficult-to-diagnose compilation errors. Mandating and causing evaluation of the concept to hard error would make this subtle failure mode obvious.

Note that this is a generalization of the previous suggested poll.

Accidental Error and Stopped Completions

`exit_scope_sender_in` should mandate (i.e. hard error rather than concept evaluating to `false`) that the sender has no non-value completions

A projection into the asynchronous domain of Eric Niebler's rationale described in the previous section: It's easy to accidentally forget `noexcept` or `unstoppable` and cause a sender to have error or stopped completions.

Consider the exit scope sender generated by `sync_object`:

```
just() | then([ptr]() noexcept {
  destroy_at(ptr);
})
```

This sender satisfies `exit_scope_sender` and therefore has completion signatures `completion_signatures<set_value_t()>`. However consider the following attempt:

```
just() | then([ptr]() {
  destroy_at(ptr);
})
```

Wherein the user either forgets `noexcept` or deliberately leaves it off because they believe that operations which facially do not throw may be induced to throw by some post hoc design [21]. This sender has completion signatures (ignoring template parameter order)

`completion_signatures<set_value_t(), set_error_t(exception_ptr)>` and therefore does not satisfy `exit_scope_sender`.

Heterogeneous Exit Scope Senders

Enter scope senders should permit multiple value completions each of which is associated with a different type of exit scope sender

This would generalize the design at the expense of implementation complexity (and possibly compile time). No use case for this has emerged as of yet.

Enter Scope Sender Value Completion Arity

The value completion of enter scope senders should be allowed to transmit values beyond just the exit scope sender (i.e. enter scope senders should be allowed to have value completions with arity greater than one)

Gašper Ažman suggested this approach during the Q&A section of the C++Now 2026 talk [19] which described the design contained in this paper.

`within` Parameter Order

`within` should accepts its parameters in the order `within(scope, sndr)` rather than `within(sndr, scope)`

`within(scope, sndr)` was the parameter order from R0 of this paper. That order is not compatible with `within` being a pipeable sender adaptor object (next poll).

Pipeable `within`

`within` should not be a pipeable sender adaptor object

Lewis Baker (among others) is of the opinion that `within` being a pipeable sender adaptor object is confusing because within attaches work (i.e. entering the scope) as a prefix of the piped work, which belies intuitive expectations about how piped work composes.

`within` was not pipeable in R0 of this paper (its parameter order was fundamentally incompatible therewith, see above).

`lifetime` Concept Checking

`lifetime` should perform eager concept checks on its arguments

Currently `lifetime` performs concept checks on its arguments when forming a sender. These concept checks are not necessarily reflective of the point of use (due to cv- and ref-qualification differences) but do help to disambiguate the two forms of `lifetime`.

Review History

R0

Presented to SG1 in Croydon 2026-03-24. The following polls were taken:

POLL: We like the direction of P3955r0, encourage more work in this direction, and suggest LEWG look at it (it will need to return to SG1 before it's forwarded).

SF	F	N	A	SA
6	6	2	0	0

Attendance: ?? IP, ?? online
# of Authors: 1
Author's Position: SF
Outcome: Unanimous consent

Revision History

R1

- Various wording improvements
- Addition of `lifetime` overload which accepts an enter scope sender factory
- Addition of proposed wording
- Added discussion of binary vs. variadic `_in` concepts
- Addition of proposed/desired polls
- Swapped the order of the parameters to `within`
- Reworked description of the proposed design to include more rationale and less almost-specification (since there's actual wording in this revision)

Acknowledgements

The author would like to thank Kirk Shoop for his permission to continue his exploration of this problem space.

The author would like to thank Eric Niebler, Ian Petersen, and Mark Hoemmen for feedback on this paper.

References

[1] M. Dominiak et al. std::execution P2300R10
[2] K. Shoop. async-object - aka async-RAII P2849R0
[3] K. Shoop et al. Cancellation is serendipitous-success P1677R2
[4] I. Petersen et al. async_scope - Creating scopes for non-sequential concurrency P3149R7
[5] A. Williams. Let_async_scope P3296R4
[6] P. Sommerlad et al. Generic Scope Guard and RAII Wrapper for the Standard Library P0052R10
[7] R. Leahy. Rename async_scope_token P3685R0
[8] R. Leahy. When Do You Know connect Doesn't Throw? P3388R3
[9] https://github.com/NVIDIA/stdexec/blob/485160802ee5ca42ca4915e3a2330579efae4ea3/include/exec/sequence.hpp
[10] R. Leahy. Evolving C++ Networking with Senders & Receivers (Part 2). Core C++ 2024
[11] R. Leahy Of Operation States and Their Lifetimes P3373R2
[12] R. Leahy. Towards Async Everything Part 1: Senders as the Lowest Layer. C++Now 2026
[13] D. Kühl et al. Add a Coroutine Task Type P3552R3
[14] R. Leahy. is_nothrow_connectable_in P4191R0
[15] R. Leahy. Stop the Decay P4288R0
[16] R. Leahy. std::execution::sequence P4320R0
[17] R. Leahy. variant_sender P4329R0
[18] https://github.com/NVIDIA/stdexec/pull/1790
[19] R. Leahy. Towards Async Everything Part 2: Scopes, Construction, and Destruction. C++Now 2026
[20] E. Niebler. Early Diagnostics for Sender Expressions P3164R4
[21] R. Leahy. Throwing Violation Handlers Are Post Hoc Library Design P4254R0
[22] L. Teodorescu. Primitives for Non-Local Concurrency P4215R0
[23] R. Leahy. when_all is just just() P4217R1
[24] R. Leahy. when_all Oughtn't Hallucinate set_stopped P4269R0