Towards Senders in Interfaces

Document number: P4223R0
Date: 2026-05-11
Repy-to: Ian Petersen <ispeters@gmail.com>
Audience: SG1, LEWG

Abstract

Other than `std::execution::task`, all of the standard senders are instances of exposition-only types, which means their types can only be used in deduced contexts. Furthermore, the computations represented by compositions of standard senders are fully reflected in the types of those compositions. These characteristics make the majority of standard senders unusable as the return values of separately-compiled functions, preventing their use on interface boundaries. `task` *can* serve in that role, but, as a coroutine type, it is unsuitable in many domains. This paper addresses this gap by proposing `std::execution::function`, an allocator-aware class template, specializations of which are senders that can serve as the return type of separately-compiled asynchronous functions.

Also proposed is a new query, `std::execution::get_frame_allocator`, inspired by Vinnie Falco *et al.*'s work in P4003. When `std::execution::function` needs to dynamically allocate space for an operation state, it prefers to use the allocator returned from `get_frame_allocator`, giving users the means to control the allocation strategy for asynchronous function activation frames independently from general allocations to take advantage of the distinct patterns exhibited by activation frame allocation.

Revision History

- R0 – First revision

Background

Separately-compiled functions are useful; even with modules making it cheaper at compile time to expose the implementation details of shared functions, there's good reason to declare a function in one place and define it in another while leaving implementation freedom in the definition. The existing standard senders (other than `std::execution::task`) can't serve in this role because they are all of exposition-only types, and the full structure of the computation they represent is expressed in their types, which defeats any effort to separate interface from implementation. The standard library ought to fill this gap by providing a sender that can serve as the return type of a function with separate declaration and definition, including virtual member functions.

Requirements

The requirements of a solution are:

- **Separation of interface and implementation**; the primary need identified in this paper is the ability to separate *what* an async function does (its interface) from *how* it does it (its implementation). The problem with the existing senders is that their types reflect the fully composed computation they represent so it is impossible to achieve this separation with the status quo.
- **Broadly applicable**; senders are intended to be usable everywhere that C++ can be used so the means for separating interface from implementation should be just as widely usable. Meeting this requirement includes customizable control over any memory allocation that is required, and the ability to avoid throwing exceptions.
- **Fully general**; there must be no arbitrary limitations on the async functions that can be abstracted. In practice, this requires supporting arbitrary combinations of completion signatures, and arbitrary combinations of environment queries.
- **Composable with other senders**; separating an async function's interface from its implementation must not render that function incompatible with the surrounding async C++ ecosystem. The most direct way to meet this requirement is to effect the interface-implementation separation through a type that models `sender`.
- **Ergonomic interface**; interface boundaries exist at all layers of software so describing those boundaries is a common activity. The mechanism should therefore be straightforward to use, easy to understand, and have good defaults.

Designs Considered

`exec::any_sender`

`exec::any_sender` (and `unifex::any_sender_of` before it) provides a type-erased container that can store an arbitrary sender that matches the constraints specified by the arguments provided to the class template. For example (https://godbolt.org/z/W5oYj4a6d),

```
// async_int.h
#include <exec/any_sender_of.hpp>
#include <stdexec/execution.hpp>

using any_int_receiver = exec::any_receiver<
   STDEXEC::completion_signatures<STDEXEC::set_value_t(int)>>;
using any_int_sender = exec::any_sender<any_int_receiver>;

any_int_sender async_int(int i);
```

```cpp
// async_int.cpp
#include "async_int.h"

namespace ex = STDEXEC;

any_int_sender async_int(int i) { return ex::just(i); }

// main.cpp
#include <stdexec/execution.hpp>

#include "async_int.h"

namespace ex = STDEXEC;

int main() {
   auto [result] = ex::sync_wait(async_int(42)).value();

   return result;
}
```

The design of `any_sender` results in three type-erased entities: the sender, the receiver to which it is connected, and the operation state that results therefrom. The type-erased receiver is of fixed size because it can be implemented in terms of a pointer to the concrete receiver plus a vtable, but the type-erased sender and the operation state that results from connecting it to a type-erased receiver can both be of arbitrary size so using an `any_sender` may require two allocations: an early one to make space for the sender, and a late one to make space for the operation state.

exec::function

`exec::function` goes in a different direction; rather than erasing an arbitrary sender, `function` erases a sender *factory* and bundles it with the arguments to that factory. Usage looks like this (https://godbolt.org/z/Gs1Tb1Tsj):

```cpp
// async_int.h
#include <exec/function.hpp>
#include <stdexec/execution.hpp>
```

```
exec::function<int(int) noexcept> async_int(int i);

// async_int.cpp
#include "async_int.h"

namespace ex = STDEXEC;

exec::function<int(int) noexcept> async_int(int i) {
   return exec::function<int(int) noexcept>(
       std::move(i),  //
       [](int i) noexcept { return ex::just(i); });
}

// main.cpp
#include <stdexec/execution.hpp>

#include "async_int.h"

namespace ex = STDEXEC;

int main() {
   auto [result] = ex::sync_wait(async_int(42)).value();

   return result;
}
```

Like the existing standard senders, `function` has no required allocation upon construction because it merely curries its arguments into a tuple. When a `function` is connected to a suitable receiver, the curried arguments are passed to the type-erased sender factory resulting in a sender of arbitrary size; that sender is connected to a type-erased receiver that, like with `any_sender`, is of fixed size, producing an operation state of arbitrary size. Thus, `function`, like `any_sender`, may need to dynamically allocate storage for the operation state.

Feature Comparison

The following table compares the implementation details of the two considered types.

Property	`any_sender`	`function`
What is type-erased?	A sender	A sender factory
What is exposed?	Possible completions Required queries	Possible completions Required queries Sender factory parameters
May allocate upon construction?	Yes	No
May allocate in `connect`?	Yes	Yes

Requirements Comparison

The following table compares the two types for how well they meet the given requirements.

Requirement	`any_sender`	`function`
Separation of interface and implementation	✅ yes	✅ yes
Broadly applicable	🟡 maybe	🟡 maybe
Fully general	✅ yes	✅ yes
Composable with other senders	✅ yes	✅ yes
Ergonomic interface	🟡 maybe	✅ yes

The yeses

- **Separation of interface and implementation**; both designs support this separation of concerns, but with different trade-offs. `any_sender` fully erases an arbitrary sender, exposing only the details of the required sender contract (i.e. the sender's potential completions, and the set of required queries), while `function` erases a sender *factory*, which eliminates the possibility of allocating storage for a sender, but adds to its interface the parameters (type, number, and order) necessary to construct the sender.

- **Fully general**; the current implementations of `any_sender` and `function` in stdexec have slightly different interfaces, but there's no technical reason they couldn't be unified. The full extent of the sender contract can be expressed in type arguments to template parameters (with `completion_signatures` to express completions, and a newly-invented `queries` type-list class template to communicate required queries).

- **Composable with other senders**; both solutions solve the problem of separating interface from implementation with a class template, specializations of which model

`sender` so both approaches participate fully in the existing asynchronous C++ ecosystem.

The maybes

- **Broadly applicable**; both `any_sender` and `function` may have to dynamically allocate memory if the objects they type-erase are bigger than any implementation-defined small-object buffer they might employ.

  Dynamic allocation (and its attendant risk of `bad_alloc` exceptions) may exclude either approach from certain constrained environments where C++ is used. First-class allocator support can broaden the appeal, though, and both approaches can have first-class allocator support.

- **Ergonomic interface**; `any_sender` and `function` can both support equivalent interfaces for describing the senders they erase (i.e. the possible completions and required queries), but they differ in when they may allocate. Both `any_sender` and `function` may allocate in `connect`, at which point there is a receiver environment available from which an allocator can be queried, but only `any_sender` allocates upon construction.

  As Vinnie Falco *et al*. explore in P4003R2, P4172R0, and P4127R0, construction-time allocation is difficult to customize with an ergonomic interface. The two choices presented in P4172R0, section 8 are 1) via the parameter list with `allocator_arg_t`, or 2) out-of-band propagation via free functions that use hidden state to keep track of the allocator. In contrast, `connect`-time allocation can be controlled with a receiver query providing an allocator intended to be used when allocating activation frames (i.e. operation states) for asynchronous functions.

  Since `any_sender` and `function` both may allocate in `connect`, they can make equivalent use of a receiver query to retrieve the controlling allocator, but `any_sender` requires a construction-time allocator that will be, comparatively speaking, awkward to provide and that isn't needed by `function`.

Analysis

As Robert Leahy has explained, function invocation can be broken down into four steps:

1. Prepare the arguments
2. Pass the arguments
3. Allow the function to run
4. Collect the result

Unlike synchronous functions, where the final three steps are inextricably coupled, the Sender/Receiver design makes each of the above steps separately identifiable for asynchronous functions:

1. the arguments are “prepared” by currying them into a sender;
2. the arguments are “passed” by connecting the sender to a receiver, producing an operation state;
3. the function is “allowed to run” by invoking `start` on the operation state; and
4. the result is “collected” in the receiver’s completion methods (`set_value`, `set_error`, or `set_stopped`).

This decomposition of asynchronous function invocation into separable steps makes it clear that a sender is, as Robert says, “the asynchronous analogue of a nullary invocable.” It also highlights the analogy between an operation state and an “activation frame”, which is the private storage allocated for the duration of a function’s execution that is commonly called a “stack frame” when the function is a traditional synchronous one.

That `any_sender` and `function` both risk allocating when constructing their respective operation states is perhaps unsurprising in light of the analogy between asynchronous operation states and synchronous stack frames: it is the callee’s responsibility to allocate its own activation frame upon being called, not the caller’s. This responsibility lies with the callee precisely because the interface to a function is limited to its signature, which communicates parameter and return details but not the implementation detail of how much temporary storage is required during execution[1]. The allocation differences between synchronous and asynchronous functions are, of course, that the abstract machine’s definition of automatic storage (which programmers typically call “the stack”) exists only for the duration of a synchronous function invocation, which makes it unusable for asynchronous functions whose dynamic extent is not bounded by a synchronous function’s activation. The need for dynamic allocation of a type-erased operation state is thus a direct consequence of the abstract machine forcing us to use dynamic storage for asynchronous activation frames. With that said, as Vinnie Falco *et al.* articulate in section 6.3 of P4712R2, activation frame allocations follow a distinct pattern that is analogous to the allocation pattern of synchronous stack frames; as they say, “sizes repeat, lifetimes nest, and deallocation order mirrors allocation order” and an activation frame-specific allocator can exploit that pattern to minimize overhead at runtime.

In contrast to the allocation pattern of an operation’s activation frame, there is no structural reason for the allocations that may be required upon construction of `any_sender`s to follow any particular pattern because senders are value types that can be constructed in one place, passed around where they are needed, and used (or discarded) according to the needs of the program. This “extra” allocation that `any_sender` requires over `function` is therefore different

[1] This analogy can also explain the leak of implementation into interface that exists in the standard senders today: today’s fully-concrete senders are like inline functions, where the compiler must be able to see the full implementation to be able to (among other things) allocate space in the caller’s frame for the callee’s activation frame.

in kind from the activation frame allocation that they share, which is intrinsically required by any solution to the interface-implementation separation problem that admits arbitrarily large activation frames without making the size of those frames part of the interface.

The sharp divide between allocation “kinds” and the tight analogy between operation states and stack frames motivates separating the “frame allocator” out from other allocators. The desire for a means to use senders in interfaces in as broad a selection of environments as possible also motivates making frame allocation a customization point so that systems with specific allocation requirements or restrictions can exercise control over how operation states are allocated as a separate concern from allocations more generally. This motivation seems to be independent of any particular strategy for enabling senders in interfaces.

Conclusion

The abovementioned trade-offs suggest that, of the two designs considered, `function` is a better match for a broadly-applicable, ergonomic, type-erasing sender for use in asynchronous interface declarations. It is equivalent to `any_sender` in most ways, but, at the expense of erasing slightly less (i.e. exposing the sender factory parameters in the interface), imposes only the intrinsically-required amount of dynamic allocation, which broadens its applicability. This paper therefore proposes two new library extensions:

- `std::execution::function`, a class template, specializations of which are type-erasing senders, and
- `std::execution::get_frame_allocator`, a new receiver environment query whose result, when present, is an allocator to be used when dynamically allocating operation states.

In support of expressing the sender contract supported by a given specialization of `function`, this paper also proposes two new class templates, `std::execution::queries` and `std::execution::attrs`, each of which serve as type-lists to communicate the receiver queries and sender attributes, respectively, required by a given `function`.

Interface Discussion

Specifying the interface to a synchronous function is relatively straightforward:

- Specify the return type
- Specify the parameters (type and order)
- Specify whether the function may throw or not

Asynchronous functions are more complicated. They can complete on three different channels, and in multiple ways on the value and error channels. They can depend on the results of certain queries against the receiver’s environment, and they may advertise information through the sender’s attributes. Describing all of this information in a concise, readable way is non-trivial.

This paper proposes a verbose-but-flexible syntax for specifying all the details of a `function`'s interface, plus several options for specifying common interfaces with briefer syntax.

Concise Descriptions of Common Cases

The simplest case is a fallible asynchronous function from `bar` and `baz` to `int` that requires no particular receiver queries and offers no particular sender attributes:

```
function<int(bar, baz)>
```

As proposed, the foregoing type describes a sender with the completion signatures `set_value_t(int)`, `set_error_t(exception_ptr)`, and `set_stopped_t()`. The failure signal can be removed by adding `noexcept` to the function type argument:

```
function<int(bar, baz) noexcept>
```

Specifying Required Queries

Requiring that the receiver's environment and/or sender's attributes support a set of queries can be done with the `queries` and `attrs` type-list class templates, respectively, like so:

```
// require an inplace_stop_token in the receiver's environment
function<
   int(bar, baz),
   queries<inplace_stop_token(get_stop_token_t) noexcept>>

// require that the erased sender be from domain MyDomain
function<int(bar, baz), attrs<MyDomain(get_domain_t) noexcept>>

// require both
function<
   int(bar, baz),
   queries<inplace_stop_token(get_stop_token_t) noexcept>,
   attrs<MyDomain(get_domain_t) noexcept>>
```

Controlling Completion Signatures

To erase senders whose completion signatures can't be described with a function signature, such as those with multiple value or error completions, or completions with more than one parameter in the value channel, the existing `completion_signatures` class template will

serve. However, given that `function` erases a sender *factory* and thus must express the factory's parameter list in its type, we need more than just `completion_signatures`. This paper proposes that the function type syntax communicate the factory parameters, with `sender_tag` as the return type to support future extensions of `function` that can erase senders whose `sender_concept` nested alias refers to future categories of senders. The result looks like this:

```
// erase a sender that can only complete with set_value(int)
function<
   sender_tag(bar, baz),
   completion_signatures<set_value_t(int)>>

// erase a sender that can fail with an error_code but not an
exception
function<
   sender_tag(bar, baz),
   completion_signatures<
       set_value_t(int),
       set_error_t(error_code)>>

// erase a sender that can succeed in two different ways
function<
   sender_tag(bar, baz),
   completion_signatures<
       set_value_t(int),
       set_value_t(int, error_code)>>
```

Full Control

The fully-general syntax requires the user to provide the factory parameters, the completion signatures, the required receiver queries, and the required sender attributes, which can be done by combining the above variations, like so:

```
function<
   sender_tag(bar, baz),
   completion_signatures<set_value_t(int)>,
   queries<inplace_stop_token(get_stop_token_t) noexcept>
   attrs<MyDomain(get_domain_t) noexcept>>
```

Summary

A `function`'s interface includes:

- the parameters to its erased sender factory;
- the possible completions;
- the required queries on the receiver's environment; and
- the required queries in the sender's attributes.

The sender factory parameters are expressed with a function type parameter. If the function type's return type is `std::execution::sender_tag` then the sender's completions must be expressed with another parameter that is a specialization of `completion_signatures` and the function type must not be `noexcept`; otherwise, the sender's completions are computed from the function type:

- `void`-returning function types result in a `set_value_t()` completion; for all other return types `T`, the completions include `set_value_t(T)`.
- `noexcept` function types do not result in error completions; non-`noexcept` functions may complete with `set_error_t(exception_ptr)`.
- `set_stopped_t()` is in the computed completions for all function types.

The queries that are required to be valid against the eventual receiver's environment are specified with a type parameter that is a specialization of the class template `std::execution::queries`, where each parameter to `queries` is a possibly-`noexcept` function type of the form `Return(Query, Args…)`. For each such function type, the expression `q(env, args…)` must be valid, where `q` is of type `Query`, `env` is the receiver's environment, and `args…` is a pack of type `Args…`. If the function type is `noexcept` then that expression must also be `noexcept`.

Similar to receiver queries, `future` can express required attributes in the erased sender with a type parameter that is a specialization of the class template `std::execution::attrs`, where each parameter to `attrs` is a function type with the same semantics as the parameters to `queries`, except applied to the sender's attributes rather than the receiver's environment.

Proposed

`std::execution::get_frame_allocator`

Add a new subsection to the end of [exec.queries] named [exec.get.frame.allocator]:

33.5.? **`execution::get_frame_allocator`**

1. `get_frame_allocator` asks a queryable object for its associated frame allocator.

2. The name get_frame_allocator denotes a query object. For a subexpression env,
   `get_frame_allocator(env)`
   is expression-equivalent to
   MANDATE-NOTHROW(*AS-CONST*(env).query(get_frame_allocator))
   Mandates: If the expression above is well-formed, its type satisfies *simple-allocator* ([allocator.requirements.general]).
3. `forwarding_query(execution::get_frame_allocator)` is a core constant expression and has value `true`.

std::execution::function

Add a new subsection to the end of [exec.snd] named [exec.function]:

33.9.? **execution::function**

TBD, but, in code, it would look like this:

```
namespace std::execution {

template <bool Nothrow, class Return, class Query, class... Args>
struct env-archetype { // exposition only
   Return query(Query, Args&&...) const noexcept(Nothrow);
};

template <
   class Env,
   bool Nothrow,
   class Return,
   class Query,
   class... Args>
concept queryable-with-impl = // exposition only
   invocable<Query, const Env&, Args...> &&
   (!Nothrow || is_nothrow_invocable_v<Query, const Env&, Args...>) &&
   same_as<Return, invoke_result_t<Query, const Env&, Args...>>;

template <bool Nothrow, class Return, class Query, class... Args>
concept queryable-with = // exposition only
   queryable-with-impl<env-archetype<Nothrow, Return, Query, Args...>,
```

```
                        Nothrow, Return, Query, Args...>;

template <class T>
inline constexpr bool is-query-sig-v = false; // exposition only

template <class Return, class Query, class... Args>
inline constexpr bool
   is-query-sig-v<Return(Query, Args...)> = // exposition only
   queryable-with<false, Return, Query, Args...>;

template <class Return, class Query, class... Args>
inline constexpr bool
   is-query-sig-v<Return(Query, Args...) noexcept> = // exposition only
   queryable-with<true, Return, Query, Args...>;

template <class Env, class Sig>
inline constexpr bool has-query-sig-v = false; // exposition only

template <class Env, class Return, class Query, class... Args>
inline constexpr bool
   has-query-sig-v<Env, Return(Query, Args...)> = // exposition only
   queryable-with-impl<Env, false, Return, Query, Args...>;

template <class Env, class Return, class Query, class... Args>
inline constexpr bool
   has-query-sig-v<Env, Return(Query, Args...) noexcept> = // exposition
only
   queryable-with-impl<Env, true, Return, Query, Args...>;

template <class Env, class... Sigs>
inline constexpr bool has-query-sigs-v = // exposition only
   (has-query-sig-v<Env, Sigs> && ...);

template <class Sndr, class... Queries>
concept sender-attributes-contain = // exposition only
   has-query-sigs-v<env_of_t<Sndr>, Queries...>;
```

```
template <class... Queries>
   requires(is-query-sig-v<Queries> && ...)
struct queries;

template <class... Attrs>
   requires(is-query-sig-v<Attrs> && ...)
struct attrs;

template <class Rcvr>
struct opstate { // exposition only
   using operation_state_concept = operation_state_tag;

   template <class Factory>
   constexpr explicit opstate(Rcvr rcvr, Factory factory); // exposition
only

   constexpr void start() & noexcept;
};

template <class... Queries>
struct virt-env { // exposition only
   // an appropriately noexcept-qualified Return query(Args...) for
   // each function type in Queries...
};

template <class Sigs, class... Queries>
struct virt-receiver; // exposition only

template <class... Sigs, class... Queries>
struct virt-receiver<
   completion_signatures<Sigs...>,
   Queries...> { // exposition only
   using receiver_concept = receiver_tag;

   // the set of completion functions specified in Sigs...
```

```
   constexpr virt-env<Queries...> get_env() const noexcept;
};

template <class Sndr, class Sigs, class Queries, class Attrs>
inline constexpr bool matching-sender-v = false; // exposition only

template <class Sndr, class... Sigs, class... Queries, class... Attrs>
inline constexpr bool
   matching-sender-v<
      Sndr,
      completion_signatures<Sigs...>,
      queries<Queries...>,
      attrs<Attrs...>> = // exposition only
   sender-to<Sndr,
             virt-receiver<completion_signatures<Sigs...>, Queries...>> &&
   sender-attributes-contain<Sndr, Attrs...>;

template <class Factory, class Sigs, class Queries, class Attrs, class...
Args>
concept matching-sender-factory = // exposition only
   invocable<Factory, Args...> &&
   is_trivially_copyable_v<Factory> &&
   matching-sender-v<
      invoke_result_t<Factory, Args...>,
      Sigs,
      Queries,
      Attrs>;

template <class...>
struct func-impl; // exposition only

template <
   class Concept,
   class... Args,
   class... Sigs,
```

```
    class... Queries,
    class... Attrs>
struct func-impl<
    Concept(Args...),
    completion_signatures<Sigs...>,
    queries<Queries...>,
    attrs<Attrs...>> { // exposition only
    using sender_concept = Concept;

    template <
        matching-sender-factory<
            completion_signatures<Sigs...>,
            queries<Queries...>,
            attrs<Attrs...>,
            Args...>
        Factory>
         requires(!same_as<func-impl, decay_t<Factory>>)
    constexpr explicit
        func-impl(Args&&... args, Factory fact) // exposition only
         noexcept((is_nothrow_constructible_v<decay_t<Args>, Args> && ...));

    template <class Sndr, class... Env>
         requires(sizeof...(Env) == 0 && sizeof...(Queries) == 0) ||
                 (sizeof...(Env) == 1 && has-query-sigs-v<Env...,
Queries...>)
    static consteval auto get_completion_signatures();

    template <class Receiver>
    opstate<Receiver> connect(Receiver rcvr) &&;

    template <class Receiver>
         requires copy_constructible<_func_impl>
    opstate<Receiver> connect(Receiver rcvr) const& {
         return func-impl(*this).connect(std::move(rcvr));
    }
```

```
  private:
   product-type<Args...> args_;  // exposition only
   // chosen to be large enough to store one of:
   //  - pointer-to-function
   //  - pointer-to-member function
   //  - pointer-to-member data
   static constexpr size_t sender-factory-size =
       2 * sizeof(void*);  // exposition only
   byte sender-factory-storage_[sender-factory-size]{}; // exposition
                                                        // only


   using virt-receiver-t = virt-receiver<completion_signatures<Sigs...>,
                                         Queries...>;  // exposition only
   virt-opstate (*operation-factory_)(void*, virt-receiver-t,
                                      Args&&...);  // exposition only
};

template <class T, bool Nothrow>
struct completions-of-sig-impl; // exposition only

template <class completions-of-sig-impl, class... Args>
struct completions-of-sig-impl<Return(Args...), false> { // exposition
only
   using type =
       completion_signatures<set_value_t(Return),
                             set_error_t(exception_ptr),
                             set_stopped_t()>;
};

template <class... Args>
struct completions-of-sig-impl<void(Args...), false> { // exposition only
   using type =
       completion_signatures<set_value_t(),
                             set_error_t(exception_ptr),
                             set_stopped_t()>;
```

```
};

template <class Return, class... Args>
struct completions-of-sig-impl<Return(Args...), true> { // exposition only
   using type = completion_signatures<set_value_t(Return),
set_stopped_t()>;
};

template <class... Args>
struct completions-of-sig-impl<void(Args...), true> { // exposition only
   using type = completion_signatures<set_value_t(),
                                      set_stopped_t()>;
};

template <class T>
struct completions-of-sig; // exposition only

template <class Return, class... Args>
struct completions-of-sig<Return(Args...)> // exposition only
   : completions-of-sig-impl<Return(Args...), false> {};

template <class Return, class... Args>
struct completions-of-sig<Return(Args...) noexcept> // exposition only
   : completions-of-sig-impl<Return(Args...), true> {};

template <class T>
using completions-of-sig-t = // exposition only
   completions-of-sig<T>::type;

template <class...>
struct make-func; // exposition only

template <class Return, class... Args>
struct make-func<Return(Args...)> { // exposition only
   using type =
       func-impl<sender_tag(Args...),
```

```
                completions-of-sig-t<Return(Args...)>,
                queries<>, attrs<>>;
};

template <class Return, class... Args>
struct make-func<Return(Args...) noexcept> { // exposition only
   using type = func-impl<sender_tag(Args...),
                          completions-of-sig-t<Return(Args...) noexcept>,
                          queries<>, attrs<>>;
};

template <class Return, class... Args, class... Queries>
struct make-func<Return(Args...), queries<Queries...>> { // exposition
only
   using type =
       func-impl<sender_tag(Args...),
                 completions-of-sig-t<Return(Args...)>,
                 queries<Queries...>, attrs<>>;
};

template <class Return, class... Args, class... Queries>
struct make-func<
   Return(Args...) noexcept,
   queries<Queries...>> { // exposition only
   using type = func-impl<sender_tag(Args...),
                          completions-of-sig-t<Return(Args...) noexcept>,
                          queries<Queries...>, attrs<>>;
};

template <class Return, class... Args, class... Attrs>
struct make-func<Return(Args...), attrs<Attrs...>> { // exposition only
   using type =
       func-impl<sender_tag(Args...),
                 completions-of-sig-t<Return(Args...)>,
                 queries<>, attrs<Attrs...>>;
};
```

```
template <class Return, class... Args, class... Attrs>
struct make-func<Return(Args...) noexcept, attrs<Attrs...>> { //
exposition only
   using type = func-impl<sender_tag(Args...),
                          completions-of-sig-t<Return(Args...) noexcept>,
                          queries<>, attrs<Attrs...>>;
};

template <class Return, class... Args, class... Queries, class... Attrs>
struct make-func<
   Return(Args...),
   queries<Queries...>,
   attrs<Attrs...>> { // exposition only
   using type =
       func-impl<sender_tag(Args...),
                 completions-of-sig-t<Return(Args...)>,
                 queries<Queries...>,
                 attrs<Attrs...>>;
};

template <class Return, class... Args, class... Queries, class... Attrs>
struct make-func<Return(Args...) noexcept, queries<Queries...>,
                 attrs<Attrs...>> { // exposition only
   using type = func-impl<sender_tag(Args...),
                          completions-of-sig-t<Return(Args...) noexcept>,
                          queries<Queries...>,
                          attrs<Attrs...>>;
};

template <class... Args, class... Sigs>
struct make-func<
   sender_tag(Args...),
   completion_signatures<Sigs...>> { // exposition only
   using type = func-impl<sender_tag(Args...),
                          completion_signatures<Sigs...>,
```

```
                        queries<>, attrs<>>;
};

template <class... Args, class... Sigs, class... Queries>
struct make-func<sender_tag(Args...), completion_signatures<Sigs...>,
                queries<Queries...>> { // exposition only
   using type = func-impl<sender_tag(Args...),
                        completion_signatures<Sigs...>,
                        queries<Queries...>,
                        attrs<>>;
};

template <class... Args, class... Sigs, class... Attrs>
struct make-func<sender_tag(Args...), completion_signatures<Sigs...>,
                attrs<Attrs...>> { // exposition only
   using type = func-impl<sender_tag(Args...),
                        completion_signatures<Sigs...>,
                        queries<>,
                        attrs<Attrs...>>;
};

template <class... Args, class... Sigs, class... Queries, class... Attrs>
struct make-func<sender_tag(Args...), completion_signatures<Sigs...>,
                queries<Queries...>, attrs<Attrs...>> { // exposition
only
   using type = func-impl<sender_tag(Args...),
                        completion_signatures<Sigs...>,
                        queries<Queries...>,
                        attrs<Attrs...>>;
};

template <class... Args>
using function = make-func<Args...>::type;

}  // namespace std::execution
```

Acknowledgements

Thank you to Vinnie Falco and the team at C++ Alliance for your efforts to explain the shortcomings of Senders and Receivers as shipped in C++26 when applied to network IO problems. This proposal is a synthesis of my understanding of the problems you've articulated and of the Sender/Receiver architecture; it wouldn't have happened without your prompting and explanations.

Thank you to Robert Leahy for repeatedly articulating so clearly and in fine detail how senders project regular old functions into the asynchronous domain. Your talks have changed how I think about senders, and your chosen vocabulary has made it easier for me to explain and reason about async C++. Thanks also for early feedback on this paper.

Finally, thanks to Eric Niebler, Ville Voutilainen, and the rest of the folks in the std::execution Discord who helped me refine the design of `function` both before and after I started writing a prototype.

References

1. P4003R2 – "A Minimal Coroutine Execution Model" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026)
2. P4172R0 – "IoAwaitable for Coroutine-Native Byte-Oriented I/O" ((Vinnie Falco, Steve Gerbino, Mungo Gill, 2026)
3. P4127R0 – "The Coroutine Frame Allocator Timing Problem" (Vinnie Falco, C++ Alliance Proposal Team, 2026)
4. https://www.youtube.com/watch?v=S5v7f8rjSVQ – "Evolving C++ Networking with Senders & Receivers [part 1]" (Robert Leahy, 2024)