No More `transform_sender` Workarounds

Document Number: P4354R0
Date: 2026-08-31
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LWG

Abstract

This paper proposes wording-only changes to simplify the specification of `let_value`, `let_error`, and `let_stopped`, along with future, proposed asynchronous algorithms.

Background

Standard sender factories generate senders which can be destructured, i.e. given such a sender `sndr` the following:

```
auto& [tag, data, ...children] = sndr;
```

Is expected to generate two bindings, plus a pack of bindings, being bound, respectively, to:

- The tag type indicating the algorithm the sender corresponds to (e.g. `std::execution::let_value_t`)
- Any data which the algorithm needs (e.g. the invocable provided as the sole argument to `std::execution::let_value` when it is piped)
- A pack of child senders which the algorithm composes (e.g. the operations to run in parallel in the case of `std::execution::when_all`)

This destructuring is not accidental, i.e. this is an intentional interface meant to be used by, for example, algorithm customizations.

Standard sender factories generate senders which are instances of instantiations of the *`basic-sender`* exposition-only class template. This sender is meant to provide wording assistance for specifying the behavior of standard asynchronous algorithms. This wording assistance operates in tandem with a class which is specialized for the “tag type” (see above) corresponding to the asynchronous algorithm being represented: *`impls-for`*.

`basic-sender``::connect`, a member function which is used to obtain an operation state and thereby to allow the represented asynchronous operation to be realized, simply constructs and

returns an instance of an instantiation of *basic-operation*, another exposition-only class template.

basic-operation, in turn, transparently connects each child of the input *basic-sender*. In addition to this behavior, which the current wording does not allow to be customized or overridden on an algorithm-by-algorithm basis, constructing *basic-operation* also calls *impls-for*::*get-state* to populate a subobject of the *basic-operation* with customizable per algorithm state.

Discussion

Transparently connecting child senders to form child operation states is convenient for wording algorithms that, when connected, need to connect all their children, and thereafter do not wish to reuse the storage for the operation states of their child operations. This describes standard asynchronous algorithms such as `std::execution::then` and `::when_all`.

There are other sorts of algorithms, however. `std::execution::let_value`, `::let_error`, and `::let_stopped` wish to eagerly connect their child sender, but wish to reuse the storage for the operation state of that child [1]. This example is particularly interesting because pre-P2300 implementations of the aforementioned algorithms reused the storage of the child operation state, but as originally adopted P2300 did not have this behavior, bloating the size of its operation state exactly because the *basic-operation* abstraction made it difficult to customize the manner in which children are connected (this is discussed in P3373).

To achieve the behavior described above `std::execution::let_value`, `::let_error`, and `::let_stopped` use a grotesque wording strategy. Their sender factories return:

make-sender(tag, fn, sndr)

Which places the child sender in the appropriate spot for the destructuring interface described in the preceding section. However this also opts into management of the corresponding operation state via *basic-operation*. To avoid this each of `std::execution::let_value`, `::let_error`, and `::let_stopped` have a corresponding dummy tag type which is referred to in the wording as *let-tag* (note that this exposition-only name denotes a different tag for each of the algorithms). `std::execution::let_value_t`, `::let_error_t`, and `::let_stopped_t` then define a `transform_sender` member function which performs:

```
auto& [_, fn, child] = sndr;
return make-sender(
  let-tag,
  let-data{
    std::forward_like<decltype(sndr)>(child),
    std::forward_like<decltype(sndr)>(fn)});
```

This causes senders to be “customized” resulting in a new sender which does not destructure the child sender as part of the trailing pack, and therefore opts it out of the undesired functionality of *basic-operation*.

The above is confusing for readers of the standard, is inconvenient to write, and not only complicates implementations that attempt to cling closely to the code-as-wording contained in `std::execution`, but also adds needless copies or moves thereto.

An alternate strategy would be to cause *basic-sender*`::connect` to defer to a new *impls-for*`::`*connect* which defaults to behaving in exactly the way *basic-sender*`::connect` behaves today, but which can be customized by an algorithm such that basic-operation is never used, and the algorithm specifies its operation state directly.

Note that the above not only simplifies the wording of `std::execution::let_value`, `::let_error`, and `::let_stopped`, but will also simplify the wording of future, proposed algorithms [2][3].

Proposal

[exec.snd.expos]

[...]

```
namespace std::execution {
  struct default-impls {                                        // exposition only
    static constexpr auto get-env = see below;                  // exposition only
    static constexpr auto get-state = see below;                // exposition only
    static constexpr auto start = see below;                    // exposition only
    static constexpr auto complete = see below;                 // exposition only
    static constexpr auto connect = see below;                  // exposition only

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

  template<class Tag>
  struct impls-for : default-impls {};                          // exposition only
}
```

The member *default-impls*`::`*get-env* is initialized with a callable object equivalent to the following lambda:

```
[](auto, auto&, const auto& rcvr) noexcept -> decltype(auto) {
  return FWD-ENV(get_env(rcvr));
}
```

The member *default-impls*::*get-state* is initialized with a callable object equivalent to the following lambda:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) noexcept ->
  decltype(auto) {
  auto& [_, data, ...child] = sndr;
  return allocator-aware-forward(std::forward_like<Sndr>(data), rcvr);
}
```

The member *default-impls*::*start* is initialized with a callable object equivalent to the following lambda:

```
[](auto&, auto&, auto&... ops) noexcept -> void {
  (execution::start(ops), ...);
}
```

The member *default-impls*::*complete* is initialized with a callable object equivalent to the following lambda:

```
[]<class Index, class Rcvr, class Tag, class... Args>(
  Index, auto& state, Rcvr& rcvr, Tag, Args&&... args) noexcept
    -> void requires callable<Tag, Rcvr, Args...> {
  static_assert(Index::value == 0);
  Tag()(std::move(rcvr), std::forward<Args>(args)...);
}
```

The member *default-impls*::*connect* is initialized with a callable object equivalent to the following lambda:

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr rcvr) noexcept(
  is_nothrow_constructible_v<basic-operation<Sndr, Rcvr>, Sndr, Rcvr>) ->
  basic-operation<Sndr, Rcvr>
{
  return {std::forward<Sndr>(sndr), std::move(rcvr)};
}
```

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

Let `Is` be the pack of integral template arguments of the `integer_sequence` specialization denoted by *indices-for*<Sndr>.

Effects: Equivalent to:

```
((void)get_completion_signatures<
  child-type<Sndr, Is>, FWD-ENV-T(Env)...>(), ...);
```

[*Note 4*: For any types `T` and `S`, and pack `E`, let `e` be the expression *impls-for*`<T>::`*check-types*`<S, E...>()`. Then exactly one of the following is true:

- `e` is ill-formed, or
- the evaluation of `e` exits with an exception, or
- `e` is a core constant expression.

When `e` is a core constant expression, the pack `S`, `E...` uniquely determines a set of completion signatures. — *end note*]

```
namespace std::execution {
  template<class Tag, class Data, class... Child>
  struct basic-sender : product-type<Tag, Data, Child...> {     // exposition only
    using sender_concept = sender_tag;
    using indices-for = index_sequence_for<Child...>;           // exposition only

    decltype(auto) get_env() const noexcept {
      auto& [_, data, ...child] = *this;
      return impls-for<Tag>::get-attrs(data, child...);
    }

    template<decays-to<basic-sender> Self, receiver Rcvr>
    auto connect(this Self&& self, Rcvr rcvr) noexcept(see below
      noexcept(impls-for<Tag>::connect(declval<Self>(), declval<Rcvr>())))
      -> basic-operation<Self, Rcvr> {
      return impls-for<Tag>::connect({
        std::forward<Self>(self), std::move(rcvr)});
    }

    template<decays-to<basic-sender> Self, class... Env>
    static constexpr auto get_completion_signatures();
  };
}
```

It is unspecified whether a specialization of *basic-sender* is an aggregate.

An expression of type *basic-sender* is usable as the initializer of a structured binding declaration ([dcl.struct.bind]).

~~The expression in the noexcept clause of the connect member function of *basic-sender* is:~~

~~is_nothrow_constructible_v<*basic-operation*<Self, Rcvr>, Self, Rcvr>~~

[...]

[exec.let]

`let_value`, `let_error`, and `let_stopped` transform a sender's value, error, and stopped completions, respectively, into a new child asynchronous operation by passing the sender's result datums to a user-specified callable, which returns a new sender that is connected and started.

For `let_value`, `let_error`, and `let_stopped`, let *`set-cpo`* be `set_value`, `set_error`, and `set_stopped`, respectively. Let the expression *`let-cpo`* be one of `let_value`, `let_error`, or `let_stopped`. ~~Let *`let-tag`* denote a unique, empty class type for each of `let_value`, `let_error`, and `let_stopped`.~~ For subexpressions `sndr` and `env`, let *`let-env`*`(sndr, env)` be expression-equivalent to the first well-formed expression below:

- *`SCHED-ENV`*`(get_completion_scheduler<`*`decayed-typeof`*`<`*`set-cpo`*`>>(`
  `get_env(sndr), `*`FWD-ENV`*`(env)))`
- *`MAKE-ENV`*`(get_domain, get_completion_domain<`*`decayed-typeof`*`<`*`set-cpo`*`>>(`
  `get_env(sndr), `*`FWD-ENV`*`(env)))`
- `(void(sndr), env<>{})`

The names `let_value`, `let_error`, and `let_stopped` denote pipeable sender adaptor objects. For subexpressions `sndr` and `f`, let F be the decayed type of `f`. If `decltype((sndr))` does not satisfy `sender` or if `decltype((f))` does not satisfy *`movable-value`*, the expression *`let-cpo`*`(sndr, f)` is ill-formed. If `F` does not satisfy invocable, the expression `let_stopped(sndr, f)` is ill-formed.

Otherwise, the expression *`let-cpo`*`(sndr, f)` is expression-equivalent to *`make-sender`*`(`*`let-cpo`*`, f, sndr)`.

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

```
template<class Sndr, class Fn>
struct let_data {
  Sndr sndr;     // exposition only
  Fn fn;         // exposition only
};
```

~~Then let the expression *`let-cpo`*`.transform_sender(s, es...)` be expression-equivalent to:~~

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

~~except that *s* is evaluated only once.~~

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

```
namespace std::execution {
  template<>
  struct impls-for<let-~~tag~~cpo> : default-impls {
~~    static constexpr auto get-state = see below;~~
~~    static constexpr auto start = see below;~~
    static constexpr auto connect = see below;

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

Let *receiver2* denote the following exposition-only class template:

[...]

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

```
template<class Cpo, class Sndr, class Fn, class Rcvr, class ArgsVariant,
  class OpsVariant>
struct let-state {
  using operation_state_concept = operation_state_tag;
  using env_t = decltype(
    let-env(declval<Sndr>(),
    get_env(declval<Rcvr&>())));   // exposition only
  Rcvr rcvr;                       // exposition only
  Fn fn;                           // exposition only
  env_t env;                       // exposition only
  ArgsVariant args;                // exposition only
  OpsVariant ops;                  // exposition only

  template<class Tag, class... Ts>
  constexpr void impl(~~Rcvr& rcvr,~~ Tag tag, Ts&&... ts) noexcept
  {                                // exposition only
    using args_t = decayed-tuple<Ts...>;
    using receiver_type = receiver2<Rcvr, env_t>;
    using sender_type = apply_result_t<Fn, args_t&>;
    if constexpr (is_same_v<Tag, Cpo>) {
      try {
        auto& tuple = args.template emplace<args_t>(std::forward<Ts>(ts)...);
```

```
      ops.template emplace<monostate>();
      auto&& sndr = apply(std::move(fn), tuple);
      using op_t = connect_result_t<sender_type, receiver_type>;
      auto mkop2 = [&] {
        return connect(std::forward<sender_type>(sndr),
        receiver_type{rcvr, env});
      };
      auto& op = ops.template emplace<op_t>(emplace-from{mkop2});
      start(op);
    } catch (...) {
      constexpr bool nothrow =
        is_nothrow_constructible_v<args_t, Ts...> &&
        is_nothrow_applicable_v<Fn, args_t&> &&
        noexcept(connect(
          declval<sender_type>(),
          receiver_type{rcvr, env}));
      if constexpr (!nothrow) {
        set_error(std::move(rcvr), current_exception());
      }
    }
  } else {
    tag(std::move(rcvr), std::forward<Ts>(ts)...);
  }
}

struct receiver {      // exposition only
  let-state& state;    // exposition only
  Rcvr& rcvr;          // exposition only

  using receiver_concept = receiver_tag;

  template<class... Args>
  constexpr void set_value(Args&&... args) noexcept {
    state.impl(rcvr, execution::set_value, std::forward<Args>(args)...);
  }
  template<class... Args>
  constexpr void set_error(Args&&... args) noexcept {
    state.impl(rcvr, execution::set_error, std::forward<Args>(args)...);
  }
  template<class... Args>
  constexpr void set_stopped(Args&&... args) noexcept {
    state.impl(rcvr, execution::set_stopped, std::forward<Args>(args)...);
  }
```

```
    constexpr env_of_t<const Rcvr&> get_env() const noexcept {
      return execution::get_env(state.rcvr);
    }
  };

  using op_t = connect_result_t<Sndr, receiver>;                // exposition only

  constexpr let-state(Sndr&& sndr, Fn fn, Rcvr& rcvr)           // exposition only
    : rcvr(std::move(rcvr), fn(std::move(fn)),
      env(let-env(sndr, get_env(this->rcvr~~rcvr~~))),
      ops(in_place_type<op_t>, std::forward<Sndr>(sndr),
        receiver{*this~~, rcvr~~}) {}

  void start() & noexcept {
    execution::start(get<op_t>(ops));
  }
};
```

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

```
[]<class Sndr, class Rcvr>(Sndr&& sndr, Rcvr& rcvr) requires see below {
  auto& [_, ~~data~~fn, child] = sndr;
~~  auto& [child, fn] = data;~~
  using child_t = decltype(std::forward_like<Sndr>(child));
  using fn_t = decay_t<decltype(fn)>;
  using args_variant_t = see below;
  using ops_variant_t = see below;
  using state_t = let-state<decayed-typeof<set-cpo>, child_t, fn_t, Rcvr,
                            args_variant_t, ops_variant_t>;
  return state_t(std::forward_like<Sndr>(child), std::forward_like<Sndr>(fn),
    std::move(rcvr));
}
```

Let `Sigs` be a pack of the arguments to the `completion_signatures` specialization named by `completion_signatures_of_t<`*`child-type`*`<Sndr>, `*`FWD-ENV-T`*`(env_of_t<Rcvr>)>`. Let `LetSigs` be a pack of those types in `Sigs` with a return type of *`decayed-typeof`*`<`*`set-cpo`*`>`. Let *`as-tuple`* be an alias template such that *`as-tuple`*`<Tag(Args...)>` denotes the type *`decayed-tuple`*`<Args...>`. Then `args_variant_t` denotes the type `variant<monostate,` *`as-tuple`*`<LetSigs>...>` except with duplicate types removed.

Given a type `Tag` and a pack `Args`, let *`as-sndr2`* be an alias template such that *`as-sndr2`*`<Tag(Args...)>` denotes the type *`call-result-t`*`<F, decay_t<Args>&...>`. Then `ops_variant_t` denotes the type

```
variant<monostate,
        connect_result_t<child_t, let-state::receiver>,
        connect_result_t<as-sndr2<LetSigs>, receiver2<Rcvr, env_t>>...>
```

except with duplicate types removed.

The *requires-clause* constraining the above lambda is satisfied if and only if the types `args_variant_t` and `ops2_variant_t` are well-formed.

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

```
~~[]<class State, class Rcvr>(State& state, Rcvr&) noexcept {
  start(get<typename State::op_t>(state.ops));
}~~
```

Let the subexpression `out_sndr` denote the result of the invocation *`let-cpo`*`(sndr, f)` or an object equal to such, and let the subexpression `rcvr` denote a receiver such that the expression `connect(out_sndr, rcvr)` is well-formed. The expression `connect(out_sndr, rcvr)` has undefined behavior unless it creates an asynchronous operation ([exec.async.ops]) that, when started:

[...]

Acknowledgements

The author would like to thank Ville Voutilainen and Eric Niebler for expressing support for the direction of this paper.

References

[1] R. Leahy. Of Operation States and Their Lifetimes P3373R4
[2] R. Leahy. It's Scopes All the Way Down P3955R1
[3] R. Leahy. std::execution::sequence P4320R0