P2254R0 | Executors Beyond Invocables

Jared Hoberock | jhoberock@nvidia.com

2020-11-15

1 Introduction

After extensive implementation experience targeting P0443R14 interfaces at a runtime that provides mechanisms for work submission and dependency chaining, we have identified a small, yet important generalization that provides numerous benefits. Our experience strongly suggests it will be beneficial to generalize the definition of concept `executor_of` to include executors that execute types beyond `invocable`s.

Our proposal:

- Broadens the definition of `executor_of` to permit execution of objects beyond `invocable`; this can include `receiver`s.
- Defines `executor` in terms of `executor_of` and `receiver_archetype`.
- Introduces `get_executor(typed_sender) -> executor` to query a sender's executor.

These changes yield numerous benefits, improve quality of implementations, and allow executors and schedulers to become semantically consistent. Adopting this proposal involves minor changes to P0443R14 and extend the applicability of executors to other interesting use cases.

2 Background

Executors are a low-level interface for submitting work that advertises guarantees about where and how that work is executed. Because platform resources and application needs are diverse, the executor interface must be broad enough to capture that diversity. As a consequence, executors may report errors and other signals via executor-defined channels.

Schedulers are factories for senders, which are tasks that `connect` to a `receiver` to produce an `operation_state`. Once `start` is called on that state, work is submitted for execution. `start` mandates a particular error-handling protocol. Namely, the `connect`ed `receiver` acts as a channel through which errors encountered during execution are communicated.

2.1 Semantic mismatch

The roles of executors and schedulers are clearly related: both are interfaces for submitting work. A possible interpretation of this relationship is that the expression `start(connect(schedule(a)), b))` is effectively a curried form of `execute(a, b)`. Given an object `a` that is both a `scheduler` and an `executor`, and an object `b` that is both an `invocable` and a `receiver`, a programmer will reasonably expect both expressions to yield equivalent effects. Indeed, semantic equivalence is a desirable property, but P0443R14 does not guarantee it. Nor does it provide an easy way for motivated implementors to provide such guarantees.

2.1.1 Error-handling

The first semantic mismatch between `execute` and `start` is error-handling. `execute` communicates errors via an executor-defined channel, while `start` mandates that errors be communicated to the `connect`ed `receiver`. `execute(ex,f)` admits three cases of error [P1658 Supplement].

1. At the invocation of `execute(ex,f)`
2. Between `execute(ex,f)` and the invocation of `f`

3. At the invocation of `f`

Ideally, a client of `execute(ex,f)` could intercept any of these errors via generic adaptation of `ex` or `f` and deliver them to a preferred error-handling protocol. Unfortunately, there is no way to generically intercept errors of case 2 [P1525R1]. As a consequence, P0443R14 executors cannot be general purpose substrates for clients that require a particular error-reporting protocol, including the `receiver` contract.

2.1.2 Execution

The second semantic mismatch between `execute` and `start` are guarantees of execution itself. As discussed, an executor provide guarantees about where and how execution happens by advertising its established executor properties. Introspecting these allows a programmer to reason about concerns such as on which threads the invocation of a function will execute and its forward progress. By contrast, work submitted through `start` does not provide a means for programmatic introspection. As a consequence, a programmer cannot rely on `start(connect(schedule(a), b))` being semantically equivalent to `execute(a, b)`.

3 Introducing Semantic Consistency

Incompatibility between the semantics of `execute` and `start` arise from the inability of `execute` to fulfill the receiver contract and the inability of `sender`s to communicate details of execution. We propose to address this mismatch by broadening the applicability of `execute`, `executor_of`, and `executor`, and by requiring introspection of a `typed_sender`'s `executor`.

3.1 Changes to executors

3.1.1 Customization point `execution::execute`

P0443R14 2.2.3.4 p2 specifies that `execution::execute(ex,f)` require its `f` parameter to satisfy `invocable`. We propose to remove this requirement to parameterize `execution::execute` on `receiver`s and other kinds of objects beyond `invocable` while requiring executors to invoke `invocable`s and fulfill `receiver` contracts.

3.1.2 Concept `execution::executor_of<E,F>`

We propose the following changes:

- P0443R14 2.2.9 requires `F` to satisfy `invocable<remove_cvref_t<F>&>`. Remove this requirement to parameterize `execution::executor_of<E,F>` on other kinds of objects beyond `invocable`.
- Given an object `ex` and an `invocable` `f`, `execution::execute(ex,f)` invokes `f` on an execution agent created by `ex`.
- Given an object `ex` and a `receiver` `r`, `execution::execute(ex,r)` either calls `execution::set_value(r)` on an execution agent created by `ex` or otherwise fulfills the receiver contract to `r`.
- Given an object `ex` and an object `obj` which satisfies neither `invocable` nor `receiver`, `execution::execute(ex, obj)` has executor-defined semantics.

3.1.3 Concept `execution::executor<E>`

Introduce `receiver_archetype` analogous to `invocable_archetype` and define `execution::executor<E>` as an alias of `execution::executor_of<E,receiver_archetype>`. This defines `executor` as a term of art for executors providing a standard error-reporting protocol via the receiver contract. With these changes, `invocable_archetype` is superfluous.

3.2 Changes to senders

3.2.1 Customization point `execution::get_executor`

We propose a new customization point `get_executor(obj)` which returns an `executor` associated with `obj`.

3.2.2 Concept `execution::typed_sender`

We propose introducing the additional requirement

```
requires(S&& s) {
  execution::get_executor((S&&)s) -> execution::executor;
}
```

Given a `scheduler` `sched` and `receiver` `r`,

```
typed_sender auto sender = execution::schedule(sched);
operation_state auto op = execution::connect(move(sender), move(r));
op.start();
```

If `execution::set_value(r)` is called on `r`, it is so called on an execution agent created by an `executor` equal to `execution::get_executor(sender)`.

4 Discussion

The preceding changes enable the following use cases of executors, which we have validated through extensive implementation experience. Beyond examples 4.1 and 4.2, we do not propose that any of these use cases be supported by any concrete executor types which may eventually be standardized.

4.1 Example: Executor of `receiver`

Defining `executor` in terms of `receiver` allows executing `receiver`s directly:

```
struct executor1 {
  template<receiver_of R>
  void execute(R&& r) const noexcept {
    try {
      thread([r = forward<R>(r)]
      {
        execution::set_value(move(r));
      }).detach();
    }
    catch(...) {
      execution::set_error(move(r), current_exception());
    }
  }

  auto operator<=>(const executor1&) const = default;

  constexpr static auto query(execution::blocking_t) const {
    return execution::blocking.never;
  }
};
```

```
static_assert(execution::executor<executor1>);
static_assert(execution::blocking.never == execution::blocking::static_query_v<executor1>);
```

4.2 Example: Executor of `invocable`

While still allowing `executors` of `invocable`:

```
struct executor2 {
  template<invocable F>
  void execute(F&& f) const {
    thread(forward<F>(f)).detach();
  }

  auto operator<=>(const executor2&) const = default;
};

static_assert(execution::executor_of<executor2,execution::invocable_archetype>);
```

4.3 Example: Executor of `promise`

Broadening the applicability of `executor_of` allows executors to fulfill `promise`s, a use case already present in the Standard Library:

```
struct executor3 {
  void execute(promise<void>&& p) const {
    try {
      thread([p = move(p)] {
        p.set_value();
      }.detach();
    }
    catch(...) {
      p.set_exception(current_exception());
    }
  }

  auto operator<=>(const executor3&) const = default;
};

static_assert(execution::executor_of<executor3,promise<void>>);
```

4.4 Example: Executor of process

As well as spawning processes:

```
struct program {
  const char* command;
};

struct executor4 {
  void execute(program p) const {
    system(p.command);
  }
```

```
  auto operator<=>(const executor4&) const = default;
};

static_assert(execution::executor_of<executor4,program>);
```

4.5 Example: Executor of CUDA graph

And CUDA graphs:

```
struct executor5 {
  cudaStream_t s;

  void execute(cudaGraphExec_t g) const {
    if(cudaError_t e = cudaGraphLaunch(g,s)) {
      throw e;
    }
  }

  auto operator<=>(const executor5&) const = default;
};

static_assert(execution::executor_of<executor5,cudaGraphExec_t>);
```

4.6 Example: Executor/Scheduler Semantic Consistency

Broadening `executor_of` enables semantic consistency between executors and schedulers:

```
struct my_scheduler {
  executor1 ex;

  auto operator<=>(const my_scheduler&) const = default;

  struct my_sender {
    executor1 ex;

    template<template<class...> class Tuple, template<class...> class Variant>
    using value_types = Variant<Tuple<>>;

    template<template<class...> class Variant>
    using error_types = Variant<exception_ptr>;

    static constexpr bool sends_done = true;

    executor1 get_executor() const {
      return ex;
    }

    template<receiver_of R>
    struct my_operation {
      executor1 ex;
      R r;
```

```
      void start() {
        execution::execute(ex, move(r));
      }
    };

    template<receiver_of R>
    my_operation<remove_cvref_t<R>> connect(R&& r) && {
      return {ex, forward<R>(r)};
    }
  };
};
```

And enables programmatic introspection of execution created by `typed_senders`:

```
static_assert(execution::blocking.never == execution::blocking::static_query_v<
  decltype(
    execution::get_executor(declval<my_scheduler::my_sender>())
  )
>);
```

4.7 Additional Considerations

This section highlights additional proposals others may wish to consider separately.

4.7.1 Properties

This paper's proposed changes to `executor_of` and `executor` may have repercussions within the properties system. In particular, `is_applicable_property` may require update in order to interoperate with `executor_of` and `executor` types.

4.7.2 `bulk_execute` and receivers

If this proposal is accepted, we expect to apply an analogous generalization to `execution::bulk_execute`.

4.7.3 Default implementation of `schedule`

[P2235] rightfully proposes to eliminate the default implementations of `execution::execute` and `execution::connect` in order to eliminate complicated circularity. The proposal in this paper allow `execution::schedule(sched)` to faithfully map onto `sched` when `sched` is an `executor` and enables a straightforward default implementation, as illustrated by example 4.1.6.

4.7.4 Receiver contract for `invocables`

Executor use and implementations may be made marginally more convenient by introducing default implementations of `execution::set_value`, `execution::set_error`, and `execution::set_done` for `invocables`. This would be equivalent to `receiver_of` subsuming `invocable`.

5 References

P0443R14 - A Unified Executors Proposal P1658R0 - Suggestions for Consensus on Executors P1525R1 - One-Way execute is a Poor Basis Operation P2235R0 - Disentangling schedulers and executors

6 Acknowledgements

Thanks to Michael Garland and David Olsen for feedback on this proposal.