Document number: P1063R1
Date: 2018-10-05
Reply To:
Geoff Romer (gromer@google.com)
James Dennett (jdennett@google.com)
Chandler Carruth (chandlerc@google.com)
Audience: WG21 Evolution Working Group

Core Coroutines

Making coroutines simpler, faster, and more general

Introduction

> "*C++ is a general-purpose programming language providing a direct and efficient model of hardware combined with facilities for defining lightweight and efficient abstractions.*
>
> Or terser:
>
> *C++ is a language for developing and using elegant and efficient abstractions.*"
>
> — Bjarne Stroustrup, *The C++ Programming Language (4th Edition)*

The Coroutines TS provides users with an elegant and efficient abstraction for writing asynchronous code. We mean that as both sincere praise, and as a critique: the Coroutines TS provides an abstraction, but it does not provide programmers with the facilities they need to define their own elegant and efficient abstractions. Furthermore, the TS's abstraction prioritizes the asynchronous use case in a variety of ways that prevent it from being general-purpose. It gives programmers ways of extending and reusing the asynchrony abstraction, but they remain locked into many of the design tradeoffs motivated by the original use case.

Fundamentally, the Coroutines TS does not provide a direct and efficient model of hardware[1]: the primitive objects and operations that are used to implement coroutines are hidden behind an abstraction boundary.

Nearly all of the serious issues we identified in P0973R0 are reflections of this problem:

- Programmers cannot reliably prevent coroutine-based code from allocating memory, even if they know the allocation is unnecessary, because the allocation takes place behind the abstraction boundary.
- Programmers cannot control variable-capture semantics, and can all too easily overlook them entirely, because the capture is hidden behind an interface that presents itself as a function call.
- The library bindings are extremely complex because different abstractions require the underlying primitives to be composed in different ways. The TS supports this by providing APIs for the programmer to *configure how they are composed*, rather than permitting the programmer to *write code that composes them*.
- The `co_await` keyword is an overt manifestation of the TS's preference for the asynchronous use case.

[1] We suggest as a friendly amendment that the quote should say "a direct and efficient model of *the platform*". For example, C++ templates provide a direct and efficient map of the compiler's code generation facilities, rather than of any hardware feature.

In this paper, we propose exposing a minimal set of coroutine primitives that map directly to the underlying implementation. This results in a design for coroutines that is substantially simpler and yet can efficiently support a broader range of uses.

We see no practical way of making our proposed revisions backwards-compatible with the Coroutines TS design, so they must be adopted before coroutines reach an IS (if at all). Straw polling in Jacksonville indicated a strong desire to ship coroutines in C++20, but we are uncertain as to whether our proposal can be implemented and sufficiently vetted in the time remaining. As an alternative we also present a much more minimal set of changes to the TS, which we believe are feasible in the C++20 timeframe, to address some of the concerns from P0973. Unfortunately, these alternative changes necessarily avoid addressing the fundamental issue of hidden primitives, and instead focus on adding yet more configuration options to patch use cases already known to be problematic.

Non-goals

This proposal is solely concerned with “stackless” coroutines, and does not address the kinds of problems that are solved by “stackful” coroutines. We fully support the committee’s decision to pursue stackless and stackful coroutines independently.

This proposal does not attempt to extend coroutines to be a fully general monad facility. For programmers who wish to adopt a monadic approach, both our proposal and the TS are limited to supporting *linear monads*, because they do not support copying a suspended coroutine frame, and consequently do not support nondeterministically resuming from the same state with multiple inputs.

Proposed Design

Unwrap operator syntax

We propose replacing the `co_await` keyword with an operator-like token, which we tentatively suggest spelling `[<-]` (we are very open to committee feedback on the spelling):

```
optional<string> f();
string s = [<-] f();
```

```
future<string> g();
string s = [<-] g();
```

```
expected<string> h();
string s = [<-] h();
```

Our proposed spelling is intended to suggest *unwrapping*, which we regard as the most central meaning of these expressions. Correspondingly, we propose to refer to them as *unwrap expressions* rather than “await expressions”, and we refer to the operand of an unwrap expression as a *wrapper* (and its type as a *wrapper type*).

An operator-like token has two major advantages over an English-derived keyword:

- An operator can more easily avoid tying itself to a particular use case, as `co_await` is tied to asynchrony.
- An operator need not choose between colliding with existing identifiers in user code, or being so awkwardly spelled that no existing code uses it. It must avoid colliding with existing C++ syntax, but that’s a far more manageable problem.

Option: We could also introduce a binary operator analogous to `->`, such that `x op y` is equivalent to `([<-]x).y`. This would make it easier to chain applications of the unwrap operator.

Alternative: The unwrap token could be a suffix, rather than a prefix. This has the advantage of naturally supporting chaining:

```
optional_struct[->].optional_sub_struct[->].field
```

However, this would depart from C++ convention (unary operators are generally prefixes), and could reduce readability by making the token less prominent.

Alternative: A different keyword spelling could be less use-case-specific than `co_await`. However, any keyword will still suffer from the need to avoid collisions with identifiers, and it is doubtful if any keyword can fully capture the breadth of possible use cases. Our best suggestion along these lines would be something like `co_unwrap`, but unwrapping is not the sole meaning of this operation; just the most central one. For example, it’s a poor fit for unidirectional generators:

```
co_unwrap std::yield(foo);  // Huh?
```

Unwrap operator semantics

Consider how an expression `co_await x` is evaluated: the state of the enclosing coroutine is reified as an object, and passed to an algorithm that is controlled by the library associated with `x`. That algorithm may eventually do two things:

1. return a value to the coroutine’s caller, and
2. resume the coroutine, specifying the value of the `co_await` expression.

The first is mandatory and synchronous, whereas the second is optional and may be synchronous or asynchronous.

Notice that both suspension and resumption of the coroutine act as inversions of control; this is most obvious in the case of resumption, where control returns from an expression via a function call, rather than a `return` statement, but the initial transfer of control (from a stack frame to an algorithm that takes that frame as input) is also effectively an inversion of control. Thus, the coroutine and its caller act as dual *control flow domains*, separated by these complementary inversions of control.

We propose to allow the library to implement almost that entire algorithm directly in C++ code. Specifically, the library will define a function which takes a coroutine object and returns the value that is returned to the coroutine's caller, while arranging for the coroutine to be resumed in whatever manner is appropriate to the library. This change is motivated by the observation that C++ code is a far simpler and more general way to specify an algorithm than overloading the ~15 extension points of a fixed algorithm specified by the standard.

In the initial version of this proposal, that algorithm was expressed as a single function, `operator[<-]()`. However, requiring the library to deal with both control flow domains in a single function body tended to make that function body fairly complex, and necessitated an awkward dual return type syntax to express the types returned to the coroutine caller and to the caller of the unwrap operator. To address those shortcomings, we propose instead for the library to express the algorithm using a pair of functions:

- `coroutine_suspend` is invoked when execution of the coroutine is suspended. Its return value becomes the value returned to the caller (hence, it is expressed in the caller's control flow domain), and it typically makes some arrangement for the coroutine to be resumed. It corresponds closely to `await_suspend` in the Coroutines TS.
- `coroutine_resume` is invoked when execution of the coroutine resumes. Its return value becomes the value of the unwrap expression (hence, it is expressed in the coroutine's control flow domain), and it typically has very little logic. It corresponds closely to `await_resume` in the Coroutines TS.

These two are joined by a third function, `coroutine_return`, which determines the semantics of a `return` statement or implicit return from a coroutine.

All three functions take as input an object we will refer to as the *shared state*. This an object associated with the coroutine's return type that lives as long as the coroutine[2], can store long-lived state associated with the wrapper, and acts as a "hook" for name lookup to find the aforementioned functions. Thus, it corresponds closely to the concept of a "promise" from the Coroutines TS, but we have chosen not to adopt that term (or the names `await_suspend` and `await_resume`) in our proposal because they are too closely tied to the asynchronous use case.

Here's an example of how these functions might be used to define the semantics of a coroutine that returns `expected<T,E>`:

[2] In a subsequent revision of this paper, we intend to propose that the shared state be a sub-object of the coroutine, which will greatly simplify the library code that manages the shared state (among many other advantages).

```
template <typename T, typename E>
struct expected_shared_state {
  template <typename U, typename Continuation>
  expected<T, E> coroutine_suspend(
      const expected<U,E>& e, Continuation& continuation) {
    if (e.has_value()) {
      tail return continuation();
    } else {
      return unexpected(e.error());
    }
  }

  template <typename U>
  const U& coroutine_resume(const expected<U,E>& e) {
    return *e;
  }

  expected<T, E> coroutine_return(const T& value) {
    return value;
  }
}
```

(See below for a discussion of `tail return`)

As shown in this example, these functions can be members of the shared state (which in this case is stateless, and is being used only as a name lookup hook), but they can also be free functions that take the shared state as the first parameter. `coroutine_suspend` also takes the wrapper object (i.e. the operand of the unwrap expression), and the *continuation*[3], a compiler-generated function object which, when called, resumes the coroutine (this plays a role akin to `coroutine_handle`, but without the mandatory type erasure). `coroutine_resume` takes the object being unwrapped, and `coroutine_return` takes the operand of the `return` statement.

In many cases, the library code that resumes the coroutine is naturally in a position to determine the value of the unwrap expression that is being resumed, so it would be very convenient if the continuation took an argument specifying that value. However, we have no way to pass that value into `coroutine_resume`: we can't pass it as an additional parameter, because the compiler would have no way to determine the type of the corresponding argument when type-checking the coroutine body. Instead, the library must stash the value in either the wrapper or the shared state (see the appendix for examples, notably `yield_result_` in the generators, and `cached_result_` in the parser).

[3] We no longer propose a separate parameter representing the suspension point, because we no longer believe the performance benefits of that approach (if any) will be significant enough to justify the extra complexity.

The return type of the continuation is called the *suspension type*, because its primary role is to convey information about the suspended state of the coroutine to the library code that invoked the continuation. The suspension type is determined by the return types of every `coroutine_suspend` and `coroutine_return` call generated for the coroutine, which must be identical other than cv-qualification in order to facilitate tail call elimination. Note that in this simple example, the suspension type happens to also be the return type of the coroutine, but that is not the case in general.

Thus, the example above says that if `e` holds a `T` value, the coroutine is resumed, with `*e` as the value of the unwrap expression. This happens synchronously as part of the `[<-]` operation, so even though the coroutine returns when the `[<-]` operation returns, the effect is as if the `[<-]` operation simply returned control to the coroutine, which then returns normally. On the other hand, if `e` holds an error, that error is returned immediately to the coroutine's caller, and consequently the coroutine returns immediately, and the remainder of the coroutine is never executed.

Note that when evaluating an unwrap expression, control leaves the enclosing coroutine (without exiting any scopes) before `coroutine_suspend` overload is invoked. Consequently, if `coroutine_suspend` throws an exception, the coroutine will not be found during stack unwinding. An unwrap expression can only throw if its operand throws, or if `coroutine_resume` throws.

Note that for simplicity, the example above glosses over the issue of qualifiers on the `expected<T,E>` object: like `*e`, `[<-] e` should be mutable if and only if `e` is mutable, and should be an rvalue if and only if `e` is an rvalue. This can be accomplished via a set of four overloads (with the unwrapped return type qualified to match the parameter), and/or perfect forwarding (with the unwrapped return type computed via a metafunction such as P0847R0's `like_t`).

Extension point spelling alternatives

The names `coroutine_suspend` and `coroutine_resume` have the drawback that they do not immediately suggest a connection to the `[<-]` syntax that they implement, in the way that `operator[<-]` did. Furthermore, it is somewhat problematic for the core language to give special meaning to certain ordinary identifiers (our proposal shouldn't invalidate any existing code, but any pre-existing functions with these names may make it harder to adapt types in their namespace to work with coroutines).

In principle, `coroutine_suspend` and `coroutine_resume` could both be named `operator[<-]`, since they have different arities (drawing on the precedent of using arity to distinguish prefix from postfix overloads of `++` and `--`), but we think that would be unacceptably confusing. Introducing named tag parameters (e.g. `operator[<-](suspend_t, ...)`) would be awkward at best.

A better option might be to spell these functions `operator[<-] suspend()` and `operator[<-] resume()`. This would be a novel extension of the `operator` syntax, but a syntactically straightforward one (other than perhaps the case of taking the address of such a function, but that can easily be disallowed).

Finally, we could introduce a new syntax for the operation of resuming a continuation, i.e. `@ continuation` or `continuation@` rather than `continuation()` (where `@` is a placeholder for a token to be determined). `coroutine_resume` could then be spelled `operator@`, leaving `operator[<-]` for the suspend operation. However, it may be surprising that neither of the arguments to `operator@` is the operand of the `@` expression. Furthermore, we haven't found a suitable spelling for `@`; `[->]` offers a tempting symmetry, but it fairly cries out to be interpreted as a binary operator `x [->] continuation` that resumes the coroutine with `x` as the value of the unwrap expression, and as discussed earlier, we can't pass that value as an argument to the resume operation.

In any of the above cases, we expect that `coroutine_return` would be spelled `operator return`.

Coroutine lambdas

A fundamental distinction between a C++ coroutine and an ordinary function is that the state of a running coroutine (i.e. the coroutine frame) is effectively an object: it has storage and a lifetime, and provides operations that the program can invoke. However, the Coroutines TS does not expose coroutines as objects; instead, it creates them implicitly (via another core-language algorithm with its own extension points) and exposes only a type-erased handle.

We propose instead to make coroutines fully-fledged objects, which can be created and managed in the same way as any other object. Lambda syntax has exactly the properties we need for this purpose: it lets us define a callable object from a function body, and allows us to explicitly specify capture semantics. A coroutine lambda is distinguished from an ordinary lambda by the fact that it specifies a wrapper type rather than a return type, using the `[->]` syntax:

```
future<string> foo();
future<int> bar();
…
auto my_coroutine = [] [->] future<int> {
  int i = ([<-]foo()).size();
  return i + [<-]bar();
};
```

A coroutine lambda is much like an ordinary lambda, except that its state includes not only its captures, but also any local variables that must be preserved across suspensions. Similarly, it exposes not only a call operator for the initial invocation, but also one for resuming execution.

The call operator that begins execution takes the shared state as a parameter (with a type determined by the `shared_state_type` member of the wrapper type, e.g. `future<int>::shared_state_type` in the example above), essentially as a workaround for the lack of a viable constructor syntax for lambdas. The resumption call operator takes no arguments. As discussed earlier, the return type of the call operators is the suspension type, which is determined from the `coroutine_suspend` and `coroutine_return` overloads invoked by the coroutine.

The call operators are never `const` (in effect, coroutine lambdas are always implicitly `mutable`); in principle we could allow the user to specify or omit `mutable` as with ordinary lambdas, but in practice `mutable` would just be boilerplate, since it would only be correct and safe to omit it in cases where the coroutine has effectively no mutable stack variables, which we expect to be rare and marginal.

Coroutine lambdas cannot take parameters. This is for reasons of safety: the code in a coroutine may continue executing after the initial function call has returned (from the caller's point of view), so if any temporary values were passed to pointer or reference parameters of the coroutine, they would be left dangling. The inputs to a coroutine lambda are instead expressed via the capture group. See below for how coroutine lambdas can be used to define ordinary functions with parameters, etc.

Modeling coroutines as lambdas rather than functions has two major benefits: first, it enables library code to control the creation, usage, and destruction of coroutine frames in exactly the same way as any other object (and in particular, allows the creation of coroutine libraries that are allocation-free by construction, rather than at the whim of the optimizer). Second, the capture syntax gives programmers explicit control over capture semantics (in the Coroutines TS, capture semantics are controlled by the parameter types, but parameter types are API-visible, and so API owners are not always at liberty to change them). Use of capture syntax also leverages programmers' existing intuitions: reference and pointer inputs to a coroutine are potentially hazardous in the same way, and for the same reasons, as the reference and pointer captures of an ordinary lambda.

Note that exceptions have no special semantics inside a coroutine: any exception that isn't caught in the body of the coroutine will propagate to the caller that resumed the coroutine (which will typically be library code associated with the coroutine, so this shouldn't have any major functional effects).

Alternative: we could specify that exceptions that escape the coroutine are caught and forwarded to an extension point comparable to `coroutine_return`. This would provide some minor benefits (primarily, greater consistency in how exceptions and ordinary returns are propagated), but also some minor drawbacks: it complicates the API, and we would not be able to handle exceptions thrown from tail calls (i.e. unwrap expressions and return statements), which may be surprising, and may limit the consistency benefits.

Expository implementation

The following example illustrates how a compiler might generate equivalent C++17 code for a given coroutine. Of course, this is not how we expect coroutine compilation to actually work, but it can serve as a “reference implementation” to understand the API and behavior of coroutine objects.

Consider the following code:

```
expected<string, Err> foo(const string& s);
expected<int, Err> bar();

void f(const string& s) {
  auto coroutine = [&s] [->] expected<int, Err> {
        int i = ([<-]foo(s)).size();
        return i + [<-]bar();
      };
}
```

The compiler could implement that by generating the following code:

```
// Convenience helper shared by all coroutine implementations
template <typename T>
class __manual_lifetime {
  std::aligned_storage_t<sizeof(T), alignof(T)> storage_;

 public:
  template <typename... Args>
  void emplace(Args&&... args) {
    new (&storage_) (std::forward<Args>(args)...);
  }

  T& get() { return *reinterpret_cast<T*>(&storage_); }

  void destroy() {
    get().~T();
  }
};

// The generated type of the coroutine lambda
class _f_1 {
  using wrapped_return_type = expected<int, Err>;
  using suspension_type = expected<int, Err>;
  using shared_state_type = typename wrapped_return_type::shared_state_type;
 public:
  _f_1(const _f_1&) = delete;
```

```
_f_1(_f_1&&) = delete;
_f_1& operator=(const _f_1&) = delete;
_f_1& operator=(_f_1&&) = delete;

// Beginning of execution
suspension_type operator()(shared_state_type shared_state) {
  __shared_state.emplace(std::move(shared_state));
  __tmp_1.emplace(foo(s));
  __suspend_point = 1;
  tail return __shared_state.get().coroutine_suspend(__tmp_1.get(), *this);
}

suspension_type operator()() {
  switch (__suspend_point) {
    case 1:
      i.emplace(__shared_state.get().coroutine_resume(__tmp_1.get()).size());
      __tmp_1.destroy();
      __tmp_2.emplace(bar());
      __suspend_point = 2;
      tail return __shared_state.get().coroutine_suspend(__tmp_2.get(), *this);

    case 2:
      int __result = i.get() +
          __shared_state.get().coroutine_resume(__tmp_2.get());
      __tmp_2.destroy();
      __suspend_point = 3;
      i.destroy();
      return __shared_state.get().coroutine_return(__result);
  }
}

~_f_1() {
  switch(__suspend_point) {
    case 1:
      __tmp_1.destroy();
      break;
    case 2:
      __tmp_2.destroy();
      i.destroy();
      break;
    case 0:
    case 3:
      break;
  }
  __shared_state.destroy();
}
```

```
 private:
  // Implicitly invoked via lambda capture syntax
  _f_1(const string& s) : s(s), __suspend_point(0) {}

  __manual_lifetime<shared_state_type> __shared_state;

  size_t __suspend_point;

  // Captures
  const string& s;

  // Stack variables
  //
  // The layout of these members is purely illustrative; in practice we expect
  // the compiler to lay out this class using the same algorithms it uses to
  // lay out ordinary stack frames.
  __manual_lifetime<int> i;

  union {
    __manual_lifetime<expected<string, Err>> __tmp_1;
    __manual_lifetime<expected<int, Err>> __tmp_2;
  };
};

void f(const string& s) {
  auto coroutine = _f_1(s);
}
```

Some notes on how we present this implementation:

- The generated code must also ensure that the local-variable members are suitably destroyed, and the object is left in a destructible state, if an exception escapes the coroutine body. This logic is omitted from the above code for clarity and simplicity.
- See below for a discussion of `tail return`, and note in particular that the example code above presumes that the functions being invoked are defined in the current translation unit; otherwise ordinary `return` would be used.

Note that the coroutine transformation does not affect the existing rules for the sequenced-before relation; if an unwrap expression and some other operation are unsequenced, the latter operation may be evaluated before the unwrap expression, or it may not be evaluated until the continuation is resumed (which, of course, may never happen).

Coroutine functions

We expect that in most use cases, coroutine lambdas will not be part of public APIs; instead, they will be hidden implementation details of ordinary functions, which wrap the coroutine

lambdas to handle issues such as parameter passing/capture, lifetime management, and whether to defer initial invocation of the lambda. We propose a sugar syntax for defining such functions, In order to mitigate the associated boilerplate.

As a motivating example, consider this asynchronous function:

```
// Consumes all bytes from `connection`, and returns the number
// of bytes consumed. `connection` must remain live until the returned
// future is ready.
auto count_bytes(Connection& connection) {
  return future<int>([&] {
    return [&connection] [->] future<int> {
      int bytes_read = 0;
      vector<char> buffer(1024);
      while(!connection.done())  {
        bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
      }
      return bytes_read;
    }
  }));
}
```

The intent here is for `count_bytes` to construct and return a `future<int>` representing the result of executing the coroutine lambda's body. We expect this to be a very general pattern for all or nearly all coroutine functions (notice that one consequence of this design is that the suspension type becomes purely an implementation detail of the library).

However, the meaning of this code is obscured by some troublesome boilerplate:

- The coroutine lambda must be wrapped in an ordinary lambda, so that the constructor can control how the coroutine lambda object is allocated. (This problem could instead be addressed by having `future`'s constructor take a lazy parameter as proposed by P0927. Note that this solution cannot be adopted after coroutines have shipped in an IS: once we establish the convention that the argument expression must be a callback, we're stuck with it.)
- The wrapper type `future<int>` must be named in two places, the explicit constructor invocation and the wrapper type of the coroutine. Neither can realistically be deduced from the other (although the explicit constructor invocation might be able to omit the template argument, if `future` provides suitable deduction guides).
- The coroutine body proper is nested inside three levels of braces.

We propose a sugar syntax for defining such a coroutine function, in which the function body (including braces) is replaced by a coroutine lambda expression:

```
auto count_bytes(Connection& connection) [&connection] [->] future<int> {
  int bytes_read = 0;
```

```
  vector<char> buffer(1024);
  while(!connection.done())  {
    bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
  }
  return bytes_read;
}
```

This syntax can be specified as a pure rewrite to the form shown above: if a coroutine lambda expression `L` with wrapper type `R` appears in place of a function body, it behaves as if the function body were `{ return R([&] { return L; }; }`.

We contend that this syntax contains almost no boilerplate other than a smattering of punctuation. The additional syntactic elements not present in the Coroutines TS all have important, user-facing functional roles:

- The capture group specifies the capture semantics of the coroutine object.
- `[->]` acts as an introducer, specifying that the following block is a coroutine.
- `future<int>` specifies what kind of coroutine function this is, including its return type.

In all three cases, making these properties syntactically explicit has important advantages:

- The programmer has explicit control over capture behavior, so that for example an argument can be captured by value (for safety) even if the API is obliged to pass by reference. Symmetrically, the capture behavior is explicitly visible in the code, cueing the reader (and programmer) to possible safety or performance concerns.
- The explicit introducer enables both the reader and the compiler to immediately and reliably recognize coroutine code. This eliminates the need for a separate `co_return` syntax to cue the compiler that it's processing a coroutine.
- The programmer has explicit, local control over what kind of coroutine is being defined, even if they do not control the function signature, e.g. because they must match an existing API (In the Coroutines TS, this can be controlled only via a trait parameterized by the parameter and return types). Symmetrically, the reader can easily tell what kind of coroutine they are reading.

In order to support this pattern, most wrapper types will need a constructor taking a single generic argument, which is interpreted as a callback that returns a coroutine. It will often be important to constrain such a constructor to avoid matching arbitrary single arguments, so we propose introducing a type trait `std::is_coroutine`, which is true only for coroutine lambda types, to facilitate such constraints.

Although we do not propose it here, this design could plausibly be extended to support deduction of the explicit template argument `int` from the coroutine body. This is somewhat non-trivial because the semantics of the coroutine body technically depend on the concrete wrapper type of the coroutine, which would make ordinary return type deduction circular. However, we believe that for non-pathological wrapper types, the deduced return type will not

have a logical dependency on the template argument, and so we can perform ordinary return type deduction using an arbitrarily-chosen type argument as a placeholder, and then replace it with the result of the deduction.

It is less clear whether we can deduce the *entire* wrapper type (not just the template argument `int`, but the template `future`). We will not go into that issue here except to say that the Coroutines TS currently does not support such deduction either, and the proposed solutions that we are aware of can be applied equally well to either proposal.

Tail calls

Consider a coroutine like the following:

```
[&connection] [->] expected<int, Err> {
  int bytes_read = 0;
  vector<char> buffer(1024);
  while(!connection.done())  {
    bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
  }
  return bytes_read;
}
```

With the design described above, the unbounded iteration in this code will be transformed into an unbounded *recursion*, raising obvious concerns about stack size. However, the mutual recursion between `coroutine_suspend` and the generated coroutine code is actually all tail recursion, because every mutually recursive call is actually the final operation before the enclosing function returns. Consequently, the compiler should be able to apply tail call elimination (hereinafter “TCE”) to avoid growing the stack.

For this approach to be viable, programmers will need to have complete confidence that TCE will in fact be applied (even in non-optimizing build modes). Consequently, we propose to standardize TCE as a C++ feature. Although obviously motivated by the coroutines use case, this proposal is completely independent of coroutines.

There are two reasons TCE is difficult to achieve in C++:

- There is currently no way to specify that TCE will take place, because the C++ standard has no explicit concept of stack storage as a finite resource.
- It’s not as easy as it seems to determine whether a call is eligible for TCE in the first place. For example, a statement of the form `return f(...);` is nevertheless ineligible if there are any local variables with nontrivial destructors still live at that point (because then the function call is not actually the last operation before the return), or if the `f()` call takes a pointer or reference to any local variable. This is not an issue for `coroutine_suspend` calls inside the coroutine generated code (because the compiler can ensure that it’s able to apply TCE to the code it generates), but it is an issue when `coroutine_suspend` invokes the continuation synchronously.

To address the first issue, we propose adding standard wording such as the following:

> "If this International Standard specifies that a function invocation is a *tail call*, then before entering that invocation, the implementation must disregard the invoking function call for purposes of enforcing any implementation-defined limits concerning the number of simultaneously active function calls, or the number or size of simultaneously-live variables with automatic storage duration. [*Note:* The effect of this requirement is that on implementations with a bounded stack, a tail call must reuse the stack frame of the calling function. — *end note*]

To address the second issue, we propose introducing a new syntax `tail return`, which requires its operand to be a tail call (`tail` is a contextual keyword, with a special meaning only when followed by `return`, so this should not break any existing code). This would be both a constraint on the operand (to make it eligible for TCE) and a requirement on the implementation (to apply TCE). The standard wording would be something like the following:

> If a `return` statement is preceded by `tail`, then evaluation of its operand will be a tail call, and the program is ill-formed if:
>
> - the statement is within a *try-block* or *function-try-block*,
> - any live object with automatic storage duration within the scope of the enclosing function has a non-trivial destructor, or its address is taken or it is bound to a reference (including the implicit object parameter of a member function) anywhere within the function body,
> - the operand is not a function call expression,
> - the operand is a function call expression whose *postfix-expression* has a function pointer type,
> - the operand is a call to a virtual function that is not named by a *qualified-id*, or
> - the function designated by the function call expression is not defined in the current translation unit, or has a return type that is not the same as the return type of the calling function (ignoring cv-qualifiers), or has a *parameter-declaration-clause* that terminates with an ellipsis.

We believe that the above conditions are minimally sufficient to permit TCE in Clang, and probably in any other reasonable C++ implementation (of course, we particularly welcome implementer feedback on this point). Note that the generated code for a coroutine lambda can easily ensure that all these conditions hold for its invocations of `coroutine_suspend`, except that it cannot guarantee that the operator is defined in the current translation unit. We will therefore specify that invocation of `coroutine_suspend` by a coroutine lambda is always a tail call *if* the selected overload is defined in the current translation unit.

Note that the above rules do not permit a function invoked via operator syntax (other than an `operator()` overload) to be a tail call. This is for reasons of readability: a statement like `tail`

`return *foo();` is apt to mislead the reader into thinking that `foo()` is the tail call, rather than `operator*()`.

Possible extension: we could loosen the above rules somewhat to permit taking addresses of and forming references to local variables, but specify that the lifetime of local variables ends when the tail call begins (since we forbid nontrivial destructors, the effect of this is just that it's UB to access them after that point). However, that would make this construct less safe, since changing `return` to `tail return` could break code in ways that can't be detected at compile time.

Alternative: we could achieve the same behavior via an attribute, e.g. `[[tail_call]]`. This would be more conceptually lightweight than a new contextual keyword, correctly signalling to programmers that they can disregard this feature unless they have a specific need for it. However, an attribute might not allow us to normatively mandate TCE, which we believe is necessary.

Alternative: rather than allow users to force TCE, we could make it inherent in the API for unwrap expressions. Specifically, we could allow `coroutine_suspend` to return either the suspension type of the coroutine, or a nullary callback returning the same type as `coroutine_suspend`. The generated code would then apply a "trampoline" technique, repeatedly checking if the result is a callback, and if so invoking it to obtain a new result, until it obtains an instance of the suspension type. However, this would substantially complicate the `coroutine_suspend` API (notice for example that it makes the return type of `coroutine_suspend` self-referential), and would not have the benefit of allowing TCE in non-coroutine contexts. Note also that the library may be able to implement this technique under our existing proposal.

`constexpr`

We have not worked through this issue in detail, but we see no obstacles to allowing coroutines to be `constexpr` (and uses of them to be core constant expressions) on the same terms as ordinary functions. The sample implementation given above cannot be `constexpr` because of its use of `reinterpret_cast`, but that is only as an expository way of depicting the compiler's management of the stack frame, which we know it can do in `constexpr` code because it already does.

Alternative: Patching the TS

We believe the design presented above addresses all of our major concerns with the Coroutines TS. However, we expect that many committee members will consider this change too extensive to make in the C++20 timeframe (and we don't necessarily disagree). If WG21 is committed to

shipping Coroutines as part of C++20, it should still be possible to address some of our concerns.

We could add first-class syntactic support for non-asynchronous use cases by replacing the `co_await` keyword with an operator token such as `[<-]`. After C++20, we could still introduce such a token as a synonym for `co_await`, although of course we could no longer remove `co_await`.

We could make the coroutine kind locally explicit via some form of introducer syntax. As a straw man example:

```
auto OpenFile(const string& filename) using future_coroutine<File> {
  ...
```

This would enable us to eliminate `coroutine_traits` (and hence eliminate the need for a shared global namespace of coroutine signatures), and also allow ordinary `return` in coroutines, although `co_return` would still be necessary in cases where e.g. the return value is not implicitly convertible to the return type. We could also add a capture group to the introducer syntax, to give explicit control of capture semantics:

```
auto OpenFile(const string& filename) using future_coroutine<File> [filename] {
  ...
```

Allocation and performance

We believe the following is a consensus description of the Coroutines TS status quo:

- A conforming implementation is permitted to allocate every coroutine frame via `operator new`; neither HALO nor “suspend point simplification and elimination” is ever guaranteed to occur.
- No existing implementation reliably elides unnecessary allocations.
- Making allocation elision reliable will require ABI extensions that have not yet even been prototyped.
- It is not yet clear whether coroutine frame allocation elision will be reliable in the no-optimization modes of major compilers (after all, it is very explicitly an "optimization").
- User code can unwittingly disable HALO, e.g. by allowing the coroutine object’s address to escape the coroutine, and it’s not yet clear how we’d teach users to avoid those hazards.
- RVO currently cannot be applied to coroutine returns.

Consequently, as one example, it is impossible to write a generator function that is guaranteed not to allocate, unless you can modify the function signature in order to trigger a custom `operator new` overload. We contend that in order for coroutines to be legitimately “zero overhead” for the generator use case, it must be possible to write a generator that is guaranteed not to allocate, if the corresponding non-generator-based code is guaranteed not to allocate (and uses only a bounded amount of stack).

Similarly, it is impossible to write a function that returns `expected<T,E>` and uses `co_await` for error propagation, but is guaranteed not to allocate.

We could address these problems through the following extensions:

- Extend the coroutine promise API with a static member `is_resumable`, which specifies whether coroutines that use that promise can be resumed. This will permit types such as `expected<T,E>` to opt out of support for resumption.
- Add wording to normatively require allocation elision when `is_resumable` is false, or when the conditions for HALO apply (e.g. the relevant operations are inlineable, and the coroutine object satisfies some specified set of conditions that imply that it does not escape).
- Permit coroutines to be `constexpr`, and specify that when `is_resumable` is true, violations of the HALO conditions cause an expression to fail to be a core constant expression.
- Extend the coroutine promise API to expose the storage location where a return value should be constructed, in order to enable RVO in coroutines (we understand that Gor Nishanov is working on a specific proposal for this).

These all appear to be pure extensions, so they could be done post-C++20 if need be.

API Complexity

We see no viable way to address the API complexity of the Coroutines TS via such incremental changes. Indeed, the changes we discuss will add yet more extension points, and we think it is likely that there will be a more or less perpetual drip of new extension points and new complexity, if we proceed with the TS design. The only way we see to fundamentally simplify coroutines is to give user code direct access to the primitive objects and operations that constitute the feature. So long as the primitives are hidden behind an abstraction boundary, it will remain necessary to poke holes in that abstraction in order to meet the needs of our diverse and highly performance-sensitive user community.

Comparison

The following chart summarizes what we see as the key functional differences between the Coroutines TS status quo, the TS with incremental fixes, and our proposal:

	Coroutines TS	**Incremental alternative**	**Core coroutines**

Library customization points	15:
	
	`await_transform`
	`operator co_await`
		17:
		
		`await_transform`
		`operator co_await`
			4:
			
			`coroutine_suspend`

	`await_ready`
	`await_suspend`
	`await_resume`
	`yield_value`
	`return_value`
	`return_void`
	`promise_type`
	`get_return_object`
	`get_return_object_on_allocation_failure`
	`coroutine_traits`
	`initial_suspend`
	`final_suspend`
	`unhandled_exception`
		`await_ready`
		`await_suspend`
		`await_resume`
		`yield_value`
		`return_value`
		`return_void`
		`promise_type`
		`get_return_object`
		`get_return_object_on_allocation_failure`
		`coroutine_traits`
		`initial_suspend`
		`final_suspend`
		`unhandled_exception`
		`is_resumable`
		`return_value_slot`
			`coroutine_resume`
			`coroutine_return`
			`shared_state_type`

Coroutine object representation	Type-erased as `coroutine_handle`	Type-erased as `coroutine_handle`	Concrete object with anonymous type
Coroutine allocation (normative)	All coroutine objects are heap-allocated by default. This can be disabled by explicit collaboration between library and user code.	All coroutine objects are heap-allocated by default, but libraries can opt out. This constrains their usage to certain optimizable patterns, which seem to cover known common cases where allocation is unnecessary. Implementations are normatively required to implement the necessary optimizations.	Coroutine objects are allocated by explicit code, just like all other objects. Allocation will normally be a hidden detail of the library.
Coroutine allocation (QoI)	Optimizers have demonstrated ability to elide coroutine allocations in many common cases. Techniques sufficient to reliably elide allocation for specific types are on the drawing board. Unclear whether	Same as Coroutines TS.	Allocation elision applies equally to all kinds of objects, including coroutines.

	optimizations will apply in all build modes.		
(N)RVO in coroutines	No	Yes	No NRVO if there's a suspend point between construction and return.
Programmer control of capture	No	Yes	Yes
`return` **in coroutines**	Forbidden	Allowed, but `co_return` is still needed in some cases.	Allowed without restriction (`co_return` is unnecessary)
User-facing syntax	Keyword, concurrency-specific	Operator token, general-purpose	Operator token, general-purpose

Conclusion

C++ is a language that enables programmers to build powerful and efficient abstractions by composing simple primitives that are efficiently supported by the platform. This is a defining property of C++, and a cornerstone of its success, so we should not abandon it (or even postpone it) without extremely compelling reasons.

The current design of the Coroutines TS is not consistent with that principle, because it does not provide simple, composable primitives, but only a complex abstraction that is tuned for a particular kind of use case. Shipping the current design as part of a C++ IS would be either an outright rejection of that principle or, at best, a wholly unjustified gamble that we'll be able to add the necessary primitives as a non-breaking extension, and still end up with a coherent design.

We believe that C++ can still be a vital language 50 years from now, and the language should be designed with that goal in mind. In 50 years nobody will even remember whether coroutines shipped in C++20 or C++23, but if we lock ourselves into a coroutines design that lacks such an essential ingredient of C++'s success, the consequences could easily last that long.

We have shown that a revised design that accords with that principle is well within reach, and that the resulting facility will be simpler, more general, and more efficient. We therefore urge the committee not to merge the Coroutines TS into the IS in its current form, and instead to allow sufficient time for this design to be fleshed out and validated.

Acknowledgements

Many thanks to Richard Smith, Gor Nishanov, Roman Perepelitsa, Jeffrey Yasskin, Bryce Lelbach, Michael Spencer, Davide Italiano and Gabriel Kerneis for their valuable design discussions and feedback.

Revision History

Changes since P1063R0:

- Replaced `operator[<-]` with `coroutine_suspend` and `coroutine_resume`, thereby eliminating the novel return-type syntax, and correspondingly revised the coroutine lambda to not take the unwrapped value as an argument on resumption.
- Eliminated explicit passing of the suspend point; the coroutine generated code now tracks it internally.
- Renamed the "final return functor" to "shared state", replaced its call operator with a named function `coroutine_return`, dropped the requirement to pass it into the coroutine on every resumption, and required the wrapper to specify the shared state type rather than vice-versa.
- Revised the coroutine lambda syntax, dropping the `do` keyword and adding a mandatory syntax for specifying the wrapper type.
- Proposed a coroutine-specific sugar syntax rather than one built out of general-purpose extensions.
- Dropped dependencies on P0927R0.
- Dropped `raise()` operation on coroutine lambdas, which is superseded by `coroutine_resume`
- Fleshed out the specification of `tail return` based on implementation experience.
- Removed `no_alloc` member from alternative proposal; programmers can force allocation to be a build failure by deleting `operator new`.
- Added a parser combinator example.
- Miscellaneous copyediting, clarification, and improved exposition.

Appendix: Examples

Caveat: unless otherwise indicated, these examples are completely untested.

Futures

The following is a (very) rough implementation of a future library that supports coroutines. All types other than `promise` and `future` are hidden implementation details. This implementation leaks all shared states, in order to avoid a lot of distracting reference-counting machinery:

```
// Interface of all future shared states. This API should be sufficient
// to implement future<T>.
template <typename T>
class future_shared_state {
 public:
  virtual bool is_ready() const = 0;
  virtual T& get() const = 0;

  virtual concrete_shared_state<T>& get_concrete() = 0;

  virtual ~future_shared_state() = 0;
};

// Interface of all promises.
template <typename T>
class promise_interface {
 public:
  virtual void set_value(const T& value) = 0;
  virtual void set_exception(std::exception_ptr ptr) = 0;
  virtual ~promise_interface() = 0;
};

// Tag type representing a shared state that is not yet ready.
struct not_ready{};

// A shared state implementation for ordinary promise/future patterns.
template <typename T>
class concrete_shared_state
    : public promise_interface<T>, future_shared_state<T> {

  std::variant<not_ready, T, std::exception_ptr> state_{not_ready{}};

  std::function<void(void)> continuation_ = nullptr;
```

```
 mutable std::mutex mu_;
 mutable std::condition_variable done_;

public:
 concrete_shared_state() = default;

 bool is_ready() const override {
   std::lock_guard guard(mu_);
   return !std::holds_alternative<not_ready>(state_);
 }

 T& get() const override {
   std::lock_guard guard(mu_);
   done_.wait(guard, [&] {
                       return !std::holds_alternative<not_ready>(state_);
                     });
   return std::visit(overloaded(
       [] (not_ready) -> T& { std::abort(); },
       [] (T& t) -> T& { return t; },
       [] (std::exception_ptr ptr) -> T& { std::rethrow_exception(ptr); }),
                     state_);
 }

 concrete_shared_state<T>& get_concrete() override {
   return *this;
 }

 template <typename U, typename Coroutine>
 void fuse_to(concrete_shared_state<U>& continuation_shared_state,
              Coroutine& continuation) {
   {
     std::lock_guard guard(mu_);
     assert(continuation_ == nullptr);
     if (std::holds_alternative<not_ready>(state_)) {
       continuation_ = [&coroutine, this] {
         try {
           continuation();
         } catch (...) {
           continuation_shared_state.set_exception(std::current_exception());
         }
       };
       return;
     }
   }

   // FIXME Fix compiler to re-check tail call when instantiating
   /*tail*/ return coroutine();
```

```
  }

  void set_value(const T& value) override {
    std::function<void(void)> continuation = nullptr;
    {
      std::lock_guard guard(mu_);
      if (continuation_ == nullptr) {
        state_.template emplace<T>(value);
        done_.notify_all();
        return;
      }
      std::swap(continuation_, continuation);
    }
    continuation();
  }

  void set_exception(std::exception_ptr ptr) override {
    std::function<void(void)> continuation = nullptr;
    {
      std::lock_guard guard(mu_);
      if (!continuation_) {
        state_.template emplace<std::exception_ptr>(ptr);
        done_.notify_all();
        return;
      }
      std::swap(continuation_, continuation);
    }
    continuation();
  }
};

template <typename T>
class coroutine_shared_state_base : public future_shared_state<T> {
  concrete_shared_state<T> shared_state_;
 public:
  coroutine_shared_state_base() = default;

  template <typename U, typename Coroutine>
  void coroutine_suspend(future<U>& f, Coroutine& continuation) {
    auto& concrete = state_->get_concrete();
    // FIXME Fix compiler to re-check tail call when instantiating
    /*tail*/ return concrete.fuse_to(*this, continuation);
  }

  template <typename U>
  U& coroutine_resume(future<U>& f) {
    return f.get();
```

```
  }

  void coroutine_return(const T& value) {
    shared_state_.set_value(value);
  }

  bool is_ready() const override { return shared_state_.is_ready(); }
  T& get() const override { return shared_state_.get(); }
  concrete_shared_state<T>& get_concrete() override {
    return shared_state_;
  }

 private:
  void set_exception(std::exception_ptr e) { shared_state_.set_exception(e); }
};

// A shared state co-allocated with a coroutine. Does not implement
// promise_interface, because the value is determined by running the coroutine.
//
// We catch all exceptions when starting the coroutine, and when resuming it
// asynchronously. We do not catch exceptions when resuming synchronously,
// because that would prevent tail call elimination; instead the exception
// propagates back to the start, and/or asynchronous resumption.
// FIXME What if we synchronously resume the callee, return to here, then throw?
template <typename T, typename Coroutine>
class coroutine_shared_state : public coroutine_shared_state_base<T> {
  Coroutine coroutine_;
 public:
  template <typename F>
  coroutine_shared_state(F&& coroutine_callback)
      : coroutine_(coroutine_callback()) {}

  void run() {
    // Begin execution of the coroutine, and return the first time it
    // blocks.
    try {
      coroutine_(*this);
    } catch (...) {
      this->set_exception(std::current_exception());
    }
  }
};

template <typename T>
class promise {
  // Invariant: if two promise objects have equal shared_state_ values, they are
  // both null.
```

```
  promise_interface<T>* shared_state_;

 public:
  promise()
      : shared_state_(nullptr) {}

  promise(promise_interface<T>* shared_state)
      : shared_state_(shared_state) {}

  promise(promise&& other)
      : shared_state_(other.shared_state_) {
    other.shared_state_ = nullptr;
  }
  promise& operator=(promise&& rhs) {
    shared_state_ = rhs.shared_state_;
    rhs.shared_state_ = nullptr;
    return *this;
  }

  explicit operator bool() { return shared_state_ != nullptr; }

  void set_value(const T& value) {
    shared_state_->set_value(value);
  }

  void set_exception(std::exception_ptr ptr) {
    shared_state_->set_exception(ptr);
  }
};

template <typename T>
class future {
  // Invariant: if two future objects have equal state_ values, they are both null
  future_shared_state<T>* state_;
 public:
  using shared_state_type = coroutine_shared_state_base<T>&;

  future(const future&) = delete;
  future& operator=(const future&) = delete;

  template <typename F>
  future(F&& coroutine_callback) {
    auto state = new coroutine_shared_state<
      T, std::decay_t<decltype(coroutine_callback())>>(coroutine_callback);
    state->run();
    state_ = state;
  }
```

```
  // Public API left as exercise for reader
};
```

A typical usage, as shown earlier, could look like:

```
auto count_bytes(Connection& connection) [&connection] [->] future<int> {
  int bytes_read = 0;
  vector<char> buffer(1024);
  while (!connection.done())  {
    bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
  }
  return bytes_read;
}
```

Simple generator

This example prints the contents of a binary tree in order, using a generator:

```
struct BstNode {
  BstNode* left, right;
  string value;
};

auto Traverse(BstNode<int>* node) [node] [->] generator<string> {
  if (node == nullptr) {
    return;
  }
  [<-] Traverse(node->left);
  [<-] std::yield(node->value);
  [<-] Traverse(node->right);
}

void PrintBst(BstNode* root) {
  generator<string> g = Traverse(root);
  while (g) {
    cout << *g << endl;
    g.next();
  };
}
```

And here's the implementation that supports it:

```
namespace std {
// yield_handle represents the result of a `yield` call. It has no semantics
// of its own; semantics are provided by the overloads for specific
```

```
// generators. Thus, all generators can use the same `yield` function.
template <typename T>
struct yield_handle {
  T& value;
};

template <typename T>
yield_handle<T> yield(T& value) {
  return {value};
}

template <typename T>
yield_handle<const T> yield(const T& value) {
  return {value};
}
}  // namespace std

// The current state of a generator<T,P>. This is a hidden implementation
// detail, but it must be a namespace-scope template in order to facilitate
// deduction of T and P.
template <typename T, typename P>
struct generator_state {
  // The code to execute to resume this generator. Null if this generator
  // is done.
  std::function<generator_state()> continuation = nullptr;

  // Pointer to the currently yielded value. Null if this generator is done.
  T* value = nullptr;
}

// generator<T, P> represents a bidirectional generator, i.e. that not only
// yields values of type T, but takes arguments of type P (which become values
// of the yield expression). Yielded values are accessed by dereferencing,
// and the generator is advanced to the next yielded value by calling next().
// Like an iterator, a generator has a special past-the-end state, signifying
// the end of the generated sequence, which cannot be dereferenced or advanced.
//
// The generator<T, void> specialization (which represents a traditional
// unidirectional generator) is omitted for brevity; the differences are
// mostly obvious, but note that it could easily implement MoveIterator
// (see P0902R0).
template <typename T, typename P = void>
class generator {
  generator_state<T,P> state_;

  // Manages lifetime of the coroutine lambda. Is not accessed otherwise.
  std::unique_ptr<void, void(*)(void*)> coroutine_;
```

```
 class shared_state_type;
 std::unique_ptr<shared_state_type> shared_state_;

public:
 struct shared_state_type {
   P* yield_result_;

   // This overload defines the semantics of yielding from a generator<T,P>
   generator_state<T,P> coroutine_suspend(
       yield_handle<T>& handle, Coroutine& continuation) {
     tail return {continuation, &handle.value};
   }

   P& coroutine_resume(yield_handle<T>&) {
     return *yield_result_;
   }

   // Unwrapping a generator object behaves like python's `yield from`: next()
   // operations on the outer generator are delegated to the inner generator
   // until it is done, and then the outer generator's coroutine is resumed.
   // Consequently, it does not return a value, even for bidirectional generators.
   template <typename U, typename Q, typename Coroutine>
   generator_state<T, P> coroutine_suspend(
       generator<U, Q>&& inner_generator, Coroutine& continuation) {
     if (inner_generator) {
       return {[&] () {
           inner_generator.next(*yield_result_);
           tail return coroutine_suspend(
               std::move(inner_generator), continuation);
       },
       &*inner_generator};
     } else {
       tail return continuation();
     }
   }

   template <typename U, typename Q>
   void coroutine_resume(generator<U, Q>&& inner_generator) {}

   generator_state<T,P> coroutine_return() { return {}; }
 };

 friend void swap(generator& lhs, generator& rhs) {
   using std::swap;
   swap(lhs.state_, rhs.state_);
   swap(lhs.coroutine_, rhs.coroutine_);
```

```
  }

  // Move only
  generator(generator&& rhs) { swap(*this, rhs); }
  generator& operator=(generator&& rhs) { swap(*this, rhs); }

  // Constructs a generator which exposes the values yielded by
  // coroutine_callback().
  template <typename F>
  generator(F& coroutine_callback)
      : shared_state_(std::make_unique<shared_state_type>()) {
    using Coroutine = decltype(coroutine_callback());
    unique_ptr<Coroutine, void(*)(void*)> coroutine(
        new Coroutine(coroutine_callback()),
        +[] (void* ptr) { delete static_cast<Coroutine*>(ptr); });
    state_ = (*coroutine)(*shared_state_);
    coroutine_ = std::move(coroutine);
  }

  // Returns whether the generator is dereferenceable. False indicates
  // the end of the generated sequence.
  explicit operator bool() const { return state_.continuation != nullptr; }

  // Accessors for the currently yielded value. static_cast<bool>(*this) must
  // be true. Valid only until the following `next()` call.
  T& operator*() { return *state_.value; }
  T* operator->() { return state_.value; }

  // Advance to the next yielded value. static_cast<bool>(*this) must be true.
  void next(P& p) {
    auto continuation = std::move(state_.continuation);
    state_.continuation = nullptr;
    shared_state_->yield_result_ = &p;
    state_ = continuation();
    shared_state_->yield_result_ = nullptr;
  }
};
```

Zero-allocation generator

The above generator is comparable to generators as proposed by the Coroutines TS; in particular, it allocates every coroutine frame on the heap, which is extremely inefficient in many cases. The following example shows a generator that always stores its state on the stack, which isn't possible with the Coroutines TS (without changing the signatures of generator functions). As a consequence of storing its state on the stack, generator functions defined this way cannot recurse (i.e. the maximum generator stack depth must be statically known).

It should be possible to use similar techniques to define a generator library that supports recursion by using a side stack (i.e. at most one more allocation than the corresponding non-generator-based recursive code), but the API design of the side stack abstraction raises issues beyond the scope of this paper.

First, a usage example:

```
// Returns a generator whose output consists of the concatenated
// outputs of each generator produced by `generators`.
template <typename T, typename P>
auto flatten(stack_generator_base<stack_generator_base<T,P>>&& generators)
    [&] [->] stack_generator<T,P> {
  while (generators) {
    [<-]*generators;
    generators.next();
  }
}

// Returns a generator that iterates over the given range.
template <typename Range>
auto traverser(const Range& range)
    [&] [->] stack_generator<decltype(*begin(range)), void> {
  for (auto& element: range) {
    [<-] std::yield(element);
  }
}

// Returns a generator that yields `f(x)`, for each `x` yielded by `g`.
template <typename T, typename F>
auto transform_generator(stack_generator_base<T>&& g, F f)
    [&g, f] [->] stack_generator<decltype(f(*g)), void> {
  while (g) {
    [<-] std::yield(f(*g));
    g.next();
  }
}

// Toy example: turn a nested vector into nested generators, and then
// flatten them.
//
// Caveat: this code contains a dangling-reference bug that we did not have
// time to fix before publication.
void f(const std::vector<std::vector<int>>& vectors) {
  stack_generator<int> gen = flatten(transform_generator(
      traverser(vectors),
      [] (const std::vector<int>& vec) { return traverser(vec); }));
```

```
  while (gen) {
    // Do stuff with *gen
    gen.next();
  }
}
```

And the underlying implementation:

```
// The internal state of a stack_generator<T,P,Coroutine>
template <typename T, typename P>
struct stack_generator_state {
  // Pointer to the currently yielded value
  T* value = nullptr;

  // The generator we have recursed into, if any
  stack_generator_base<T, P>* nested_generator = nullptr;
};

// Base class of all stack_generators that take P and yield T.
// Allows us to type-erase the coroutine.
template <typename T, typename P>
class stack_generator_base {
  stack_generator_state<T,P> state_;
  P* yield_result_;
 public:
  stack_generator_base(stack_generator_base&&) = delete;
  stack_generator_base& operator=(stack_generator_base&&) = delete;

  void next(P& p) {
    yield_result_ = &p;
    (void) next_impl(p);
    yield_result_ = nullptr;
  }

  operator bool() const {
    return state_.value != nullptr;
  }
  T& operator*() { return *state_.value; }
  T* operator->() { return state_.value; }

  stack_generator_state<T,P> coroutine_suspend(
      std::yield_handle<T>& handle, Coroutine&) {
    return {handle.value, nullptr};
  }

  P& coroutine_resume(std::yield_handle<T>&) {
```

```
    return *yield_result_;
  }

  template <typename U, typename Q>
  stack_generator_state<T,P> coroutine_suspend(
      stack_generator_base<U,Q>&& inner_generator, Coroutine&) {
    return {&*inner_generator, &inner_generator};
  }

  template <typename U, typename Q>
  void coroutine_resume(stack_generator_base<U,Q>&&) {}

  stack_generator_state<T,P> coroutine_return() {
    return {};
  }

 private:
  template <typename T2, typename P2>
  friend class stack_generator_base<T2, P2>;

  // Resumes execution of the generator, and returns the new state
  virtual stack_generator_state<T,P> resume(P& p, size_t suspend_point) = 0;

  T* next_impl() {
    if (state_.nested_generator != nullptr) {
      T* value = state_.nested_generator.next_impl(p);
      if (value != nullptr) {
        return value;
      } else {
        state_.nested_generator = nullptr;
      }
    }
    assert(state_.nested_generator == nullptr);

    state_ = resume(p, suspend_point);
    return state_.value;
  }
};

template <typename T, typename P = void, typename Coroutine>
class stack_generator : public stack_generator_base<T,P> {
 public:
  using suspension_type = stack_generator_state<T,P>;

  template <typename F>
  stack_generator(F& coroutine_callback)
      : coroutine_(coroutine_callback()) {}
```

```
 private:
  Coroutine coroutine_;

  stack_generator_state<T,P> resume(P& p, size_t suspend_point) override {
    return coroutine_();
  }
};
```

Parser Combinators

Coroutine syntax can be used to produce clear and elegant recursive-descent parsers. Here's an example usage, which parses a toy arithmetic expression language, computing the expression value on the fly:

```
// expr ::= expr addop factor | factor
// addop ::= '+' | '-'
// factor ::= number | '(' expr ')'

enum class Sign { Plus, Minus };

Parser<Sign> AddOp() {
  return FirstMatch<Sign>(
      Map(Consume("+"), Sign::Plus),
      Map(Consume("-"), Sign::Minus));
}

auto Number() [] [->] Parser<int> {
  string_view input = [<-] Peek();
  int value;
  auto result = std::from_chars(input.begin(), input.end(), value);
  if (result.ec != std::errc{}) {
    [<-] Fail(result.ec);
  }
  [<-] AdvanceInput(result.ptr - input.begin());
  return value;
}

Parser<int> Factor() {
  return FirstMatch<int>(
      Number(),
      [] [->] Parser<int> {
        [<-] Consume("(");
        int result = [<-] Expression();
        [<-] Consume(")");
```

```
        return result;
      });
}

auto Expression() [] [->] Parser<int> {
  int result = [<-] Factor();
  while (!([<-] Peek()).empty()) {
    Sign sign = [<-] AddOp();
    int next = [<-] Factor();
    switch (sign) {
      case Sign::Plus:
        result += next;
        break;
      case Sign::Minus:
        result -= next;
        break;
    }
  }
  return result;
}
```

Here's the underlying `Parser` type (note that for exposition purposes, we err on the side of simplicity rather than efficiency):

```
// An intermediate state of a Parser. `value` represents the value just parsed
// (nullopt if the parse failed), and `tail` represents the unparsed suffix of
// the input.
template <typename T>
struct ParseState {
  optional<T> value;
  string_view tail;
};

template <typename T>
struct ParserReturnCallback {
  Parser<T> operator()(const T& val) { return Return(val); }
};

// A Parser<T> is essentially a function that takes a string_view, parses
// it to produce a T, and returns the result.
template <typename T>
class Parser {
  std::function<ParseState<T>(string_view)> parse_;
  std::optional<T> cached_result_;
```

```
 public:
  class shared_state_type {
    template <typename U, typename Coroutine>
    Parser<T> coroutine_suspend(Parser<U>& parser, Coroutine& continuation) {
      return [&] (string_view input) -> ParseState<T> {
        ParseState<U> state = parser.parse(input);
        if (!state.value) {
          return {nullopt, state.tail};
        }
        return continuation().parse(state.tail);
      }
    }

    template <typename U>
    U coroutine_resume(Parser<U>& parser) {
      return *cached_result_;
    }

    Parser<T> coroutine_return(const T& value) {
      return Return(value);
    }
  };

  template <typename F>
  Parser(F& coroutine_callback)
      requires std::is_coroutine_v<decltype(coroutine_callback())>
      : parse_([coroutine = coroutine_callback()] (string_view input)
               -> ParseState<T> {
          return coroutine({});
        }) {}

  template <typename F>
  Parser(F& parse)
      requires std::is_convertible_v<
                   decltype(parse(std::declval<string_view>)), ParseState<T>>
      : parse_(parse) {}

  ParseState<T> parse(string_view str) const {
    ParseState<T> result = parse_(str);
    cached_result_ = result.value;
    return result;
  }
};
```

And here are the reusable low-level parsing operations used in the above example:

```
// Parser which returns the given value, without consuming any input
template <typename T>
Parser<T> Return(const T& val) {
  return [=] (string_view input) { return val; };
}

// Parser which produces the entire remaining input as a string_view,
// without consuming any of it.
Parser<string_view> Peek() {
  return [] (string_view input) {
    return ParseState<string_view>{ input, input }
  };
}

// Parser which consumes n characters of input.
Parser<void> AdvanceInput(size_t n) {
  return [] (string_view input) {
    input.remove_prefix(n);
    return ParseState<void>{input};
  }
}

// Parser which fails without consuming any input.
Parser<void> Fail() {
  return [] (string_view input) {
    return {nullopt, input};
  }
}

namespace internal {
ParseState<T> FirstMatchImpl(string_view input) {
  return {nullopt, input};
}

template <typename T, typename... Parser_Ts>
ParseState<T> FirstMatchImpl(string_view input, Parser<T> parser,
                             Parser_Ts... parsers) {
  ParseState<T> result = parser.parse(input);
  if (result.value) {
    return result;
  } else {
    return FirstMatchImpl(input, std::move(parsers)...);
  }
}
}  // namespace internal

// Parser which parses a T value from the first of parsers... which
```

```
// succeeds. parsers... must all be Parser<T> objects.
template <typename T, typename... Parser_Ts>
Parser<T> FirstMatch(Parser_Ts... parsers) {
  return [=] (string_view input) {
    return internal::FirstMatchImpl(input, std::move(parsers)...);
  }
}

// Parser which consumes the given string value from the beginning of
// the input, or fails if it is not present.
Parser<void> Consume(string expected) {
  return [=] (string_view input) {
    if (input.starts_with(expected)) {
      input.remove_prefix(expected.size());
      return ParseState<void>{input};
    } else {
      return Fail();
    }
  };
}

// Parser which parses the same inputs as `parser`, and produces `value`.
template <typename T>
Parser<T> Map(Parser<void> parser, T value) [=] [->] Parser<T> {
  [<-] parser;
  return std::move(value);
}
```