ChatGPT

Generators and Coroutines in Rust: Towards Python-like Ergonomics

1. Ergonomics of Rust 2024 Generators vs Python's Generators and Coroutines

Rust's emerging generator syntax (`gen` blocks): Rust is introducing **generator blocks** (reserved under the keyword `gen` in the 2024 edition) to simplify writing iterators. A `gen { ... }` block allows using the `yield` keyword inside a block or function, producing an iterator-like generator value that can be looped over with `for` or collected into a container [1]. This is analogous to Python's generator functions, where any function containing a `yield` becomes a generator. For example, on Rust nightly (edition 2024), one can write:

```
#![feature(gen_blocks)]
gen fn numbers() -> i32 {
    yield 1;
    yield 2;
}
for x in numbers() {
    dbg!(x);
}
```

This code is functionally similar to a Python generator function that yields `1` and `2`. In Rust, the `gen fn` (or `gen { ... }` as an expression) creates a generator object that implements the iterator protocol, so it can be used in a `for` loop or have iterator adapters like `.collect()` called on it [2]. Under the hood, the compiler transforms the `gen` block into a state machine (much like how Rust's `async`/`await` works), eliminating the need for manual iterator state management.

Comparison to Python's generator functions: Python offers extremely ergonomic generators – one simply writes a normal function with `yield` statements, and Python handles the rest. Rust's new `gen` mechanism is conceptually similar, but it requires an explicit `gen` keyword to denote that a function or block is a generator (since Rust, being statically typed, must distinguish generators from regular functions at compile time). Once defined, using a Rust generator is nearly as straightforward as using a Python generator: you can iterate over it directly. For example, a Rust `gen` generator can be iterated with `for item in generator { ... }` just as Python allows `for item in generator: ...`. This is a significant ergonomic improvement over earlier Rust, where writing a custom iterator often meant implementing the `Iterator` trait by hand or using complex combinators. The motivation behind Rust's generator syntax is to allow a simple `yield`-based lazy sequence definition "so we can now write a simple `for` loop and still get a lazy iterator of values" instead of manual iterator implementations [3] [4].

Rust's async coroutines vs Python's coroutines: Rust already has *asynchronous coroutines* via `async fn` and `await`, which are analogous to Python's `async def` functions and `await` statements. However, Rust's `async` is targeted at concurrency (returning a `Future` that must be polled), whereas Python's `async def` also returns a coroutine that an event loop drives. Both are **stackless** coroutines: they don't create a separate call stack, but rather suspend and resume execution on the current stack. Notably, Rust's async/await is *implemented using the same underlying generator framework*—the Rust compiler transforms an `async fn` into a hidden generator that yields to the executor each time it awaits. The new Rust 2024 `gen` feature can be seen as exposing a similar generator capability directly to users for iterator construction, but with a focus on producing sequences of values (lazy iterators) rather than asynchronous tasks. In terms of ergonomics, Python's coroutines are highly integrated into the language (with simple syntax and rich support in the runtime), while Rust's approach requires more boilerplate (setting up an executor for `async`, or enabling nightly features for `gen`). That said, Rust offers stronger compile-time checks (e.g. enforcing Send/Sync requirements and lifetimes in async code), whereas Python's dynamic coroutines are checked at runtime.

Current limitations: One ergonomic gap is that Rust's generator syntax is still **unstable** as of 2025 – the `gen` keyword is reserved in the 2024 edition, but the feature is only usable on nightly Rust with `#![feature(gen_blocks)]` [2]. This means that, on stable Rust 1.85 (which defines Edition 2024), you get the keyword reservation but not the full generator functionality stabilized. Python, by contrast, has had stable generator support for decades. Another difference is that Rust's generators must specify the yield type (and possibly a return type) in their signature, whereas Python's generators are more flexible and can yield varying types (though typically one type is used). Rust's static typing makes generators more predictable (each `yield` must yield a value of the declared type), but also slightly less flexible than Python's dynamic yields.

Expressiveness: With `gen` blocks, many common patterns that Python developers enjoy (like generating sequences on the fly, or writing coroutines for cooperative multitasking) become more ergonomic in Rust. For instance, consider Python's ability to yield values in the middle of complex logic; Rust can now do the same without requiring an entire state-machine or an external crate. The RFC's example of run-length encoding shows how a `gen { ... }` block yields intermediate results multiple times within a single function, dramatically simplifying iterator code [5] [6]. Rust's strong iterators and `Iterator` trait adapters remain available on the resulting generator (you can chain `.map()`, `.filter()`, use `.sum()`, etc., directly on a `gen` generator). In summary, Rust 2024's generators narrow the ergonomics gap with Python: defining lazy sequences is nearly as straightforward as writing a loop with `yield` in Python, with the trade-off of explicit typing and the `gen` marker for clarity and safety.

2. Bidirectional Data Flow ("send"ing Values) in Generators

A powerful feature of Python's generators is their **bidirectional** nature: not only can a generator yield values out to the caller, but the caller can also **send** values into the generator. In Python, a generator's `yield` expression can receive a value sent by the caller via the generator's `.send(value)` method (PEP 342 introduced this). This enables coroutines and pipelines: for example, a Python generator can act on data fed into it interactively, not just produce values. A simple Python example:

```
def coro():
    x = yield "start"
    print("received:", x)
```

If you call `gen = coro(); next(gen)` you get `"start"`, and then `gen.send("hello")` will resume the generator, assign `"hello"` to `x` at the yield point, and continue execution.

Rust's approach to resume arguments: In Rust, achieving a similar two-way generator is more complex. The current Rust generator proposals initially focus on **one-way** iteration (generators that only yield out values). However, the underlying machinery does support bidirectional data flow. The unstable `std::ops::Generator` trait in Rust is defined with a type parameter for a **resume argument**. In fact, the `Generator` trait looks roughly like `trait Generator<R> { type Yield; type Return; fn resume(&mut self, arg: R) -> GeneratorState<Self::Yield, Self::Return>; }`. This means a generator can accept a value of type `R` each time it is resumed. On nightly Rust, one can construct a generator manually (using the experimental `yield` syntax or by implementing `Generator` for a closure) that takes an argument in its `resume()`. This is the low-level analog of Python's `send()`. For example, an experimental generator could be written as: `let mut gen = ||<u32> { let mut sum = 0; loop { let add = yield sum; sum += add; } };` and then one could call `gen.resume(5)` to send in the value 5, which the generator code captures as the result of the `yield` expression.

Design and current status: Resume arguments (i.e. the ability to `send` data into a running generator) are recognized as a powerful feature, but they add complexity to the language design. Rust language designers have debated unified coroutines that support resume arguments. An older proposal notes that "Generator resume arguments are a sought after feature" and discusses unifying generators with closures to allow resume arguments cleanly [7]. One challenge is **syntax** and **lifetime safety** – if a generator can receive data of some type, especially a reference, on each resume, the compiler needs to ensure that data does not outlive its scope or get aliased unsafely. The design notes for Rust coroutines consider, for example, whether resume arguments might need to be `'static` or how to prevent borrowing issues when a sent-in reference is stored across a yield [8] [9]. Because Rust prioritizes safety, the initial generator implementation likely omits resume arguments until these questions are resolved.

In the meantime, how can one simulate Python's `send()` in Rust? There are a few approaches:

- **Using shared state:** One can achieve a form of two-way communication by capturing a mutable reference or using an `RefCell` / `Mutex` that both the generator and caller can access. For instance, the generator could periodically read from some shared variable for input. However, this is not as elegant or explicit as Python's `send()`, and it doesn't truly pause at a yield point waiting for input; instead the generator would have to poll for changes.

- **Community crates:** The Rust ecosystem provided workarounds on stable Rust. The **generator-rs** crate (discussed in detail below) actually offers a `Generator` type with a `.send()` method for bidirectional use. Its API lets you specify a type that will be sent into the generator. For example, with generator-rs you can create a generator that takes input of type `u32` on each resume and yields out a `u32`. An excerpt from the crate's examples demonstrates this usage: they instantiate a generator with a send type `u32` and call `s.send(val)` in a loop to feed values in, collecting

outputs each time [10]. Under the hood, the crate's yield implementation returns a value inside the generator closure, similar to how Python's `yield` expression evaluates to the sent value on resumption.

- `genawaiter` **crate:** Another approach is to leverage Rust's async features to emulate send. The **genawaiter** crate takes a different route by using `async` / `await` to provide a yield-like API on stable Rust. In genawaiter, you write an async generator that yields values by `co.yield_(value).await`. While genawaiter primarily focuses on yielding out values, the crate's author and users have discussed the possibility of resume arguments (for example, by capturing the result of the `.await`). It's essentially using futures as coroutines. The advantage is that *"there's no assembly or platform-specific shenanigans under the hood. It's all built on top of* `async` / `await` *"* [11], which makes it portable. The downside is that each yield is an `.await`, meaning the generator must be polled like a future, and integrating a `send` would require careful design (likely passing the value via some shared context or an async channel).

In summary, Rust is capable of bidirectional generator communication, but the ergonomic, language-supported way to do it is still under development. Python's `send()` is a one-liner and very straightforward; achieving the same in Rust currently requires either unstable features or external crates. The Rust community clearly values this capability – as seen in RFC discussions and community libraries – so we can expect that future Rust versions or RFCs will introduce a clearer pattern. Perhaps a future edition will allow syntax like `let input = yield output;` inside a `gen fn`, meaning "yield `output` and suspend, then assign the next resume argument to `input` when resumed" – exactly mimicking Python's semantics. Until then, developers must rely on the patterns above for two-way flow in Rust generators. It's worth noting that the initial stabilization of generators might purposefully exclude resume arguments to focus on the common use-case of one-way sequences, with bidirectional support as a possible extension once it can be done in a sound and ergonomic manner.

3. The `generator-rs` Crate: Stackful Generators in Rust

While Rust's native generators are (at the moment) stackless (state-machine based) and unstable, the `generator-rs` **crate** (by Xudong Huang) takes a different approach: it implements **stackful** generators (also known as **coroutines**) on stable Rust. This crate essentially provides Python-like generators today, by managing its own stacks and context switching in user space. According to its README, `generator-rs` is a "rust stackful generator library" [12] that supports features analogous to Python's generators, including:

- **Yielding values and sending values (messages) back in:** the crate supports "basic send/yield with message support" [13], meaning you can both yield values out *and* receive values in, just like Python's `yield` / `send`. The API provides a method `Generator::send(input)` to resume a generator with a value, and inside the generator's closure, the `yield_` call returns the sent value (if any). This allows writing coroutines that respond to incoming data.

- **Cancellation and error handling:** It supports generator cancellation (you can force a generator to unwind and stop early), handling of panics inside the generator (so a panic doesn't abort the whole program by default), and even a notion of `done!()` to terminate the generator cleanly [14] [13].

- `yield_from` **support:** Much like Python's `yield from` syntax (PEP 380) for delegating to sub-generators, `generator-rs` offers a `yield_from` function. This lets one generator yield all values from another generator (or any iterator) seamlessly [13]. Under the hood, this likely means the library will keep resuming the inner iterator and yielding those values out of the outer generator, simplifying composition of generators [15].

- **Configurable stack size and scope:** Because these are stackful coroutines, you can configure the stack size for the generator's stack. The crate provides defaults (e.g. 4KB by default [16]) and allows tuning if needed. It also differentiates between *scoped* generators (which can borrow local data from the parent stack) and *static* generators (which require all captured data to be `'static`). For example, `Gn::new_scoped` allows borrowing from the current stack frame (the generator will be canceled when that scope exits), whereas `Gn::new` (now deprecated in favor of scoped versions) required `'static` captures. This distinction is important for safety – it prevents yielding references into a generator that outlive their owners.

Low-level implementation (assembly details): Achieving stackful coroutines in Rust requires what is essentially a user-space context switch. The `generator-rs` crate does this with carefully crafted platform-specific assembly. It allocates a new stack for each generator and uses assembly routines to swap CPU register context when switching between the generator and the caller. In particular, the crate defines assembly functions to: set up a new coroutine's stack, transfer control to the coroutine, and save/restore registers on yields. For example, on several architectures the core routines are: a **bootstrap** function that prepares the new stack frame and jumps into the generator's closure, and a **swap** function that swaps the register state (program counter, stack pointer, callee-saved registers, etc.) between the caller and the callee. A discussion of the RISC-V port of `generator-rs` describes it well: the `bootstrap_green_task` function "sets up arguments and then jumps to the address contained in a register, effectively starting a new task", and the `swap_registers` function *"saves the current values of a large number of registers (both general-purpose and floating-point) to memory and then restores them from another memory location"* [17]. In essence, when you call `gen.send(x)` (or initially start the generator), `generator-rs` saves the current thread context (instruction pointer, etc.), then loads the saved context of the generator's stack and jumps into it. When the generator calls `yield_`, the library's assembly does the inverse: save the generator's context and restore the caller's context, so execution returns to the caller with the yielded value.

This approach is reminiscent of how one might implement coroutines in C/C++ with `setjmp`/`longjmp` or assembly switching, or how operating systems implement threading. The crate avoids relying on OS-level threads by doing it at user level (thus the term "green threads" or "user-space threads" sometimes applies). Notably, the implementation does not use any nightly Rust features like the unstable `Generator` trait; instead it uses `unsafe` and assembly internally but presents a safe API.

Portability and performance: Because `generator-rs` works at such a low level, it needs custom code per CPU architecture and calling convention. The crate authors have included support for a range of architectures. As of the latest release, it supports x86_64 (on Linux, Windows, macOS, Fuchsia), AArch64 (ARM 64-bit on Linux, macOS, Android, Fuchsia), LoongArch64, ARMv7, RISC-V 64, PowerPC64LE, etc. [18] [19]. Each of these has its own assembly file implementing the context switch according to the platform's ABI. This broad coverage is impressive, but it also highlights the maintenance burden – if a new architecture (say, WASM or MIPS) needs support, a contributor must write the appropriate assembly. The crate's design is platform-neutral at the API level but not *implementation*-neutral; it essentially embeds a mini runtime for each target.

In terms of performance, context-switching with optimized assembly can be extremely fast. The similar **libfringe** library (an earlier Rust coroutine library) demonstrated context switches on the order of only a few nanoseconds on x86_64 [20]. The `generator-rs` crate likely achieves comparable performance, although early versions of the crate were slower than libfringe (one issue report noted an order of magnitude difference, prompting a suggestion to adopt libfringe's techniques). Still, a yield/resume in `generator-rs` is essentially a function call plus saving/restoring registers – on modern CPUs this is very cheap (much cheaper than an OS thread context switch). This means using `generator-rs` for intensive iterators or coroutines can be quite viable performance-wise. The trade-off comes in the form of increased complexity and a bit more overhead compared to compile-time state machines: for example, each generator has its own stack allocation (memory overhead) and the context switch, while fast, is not "zero cost" like inlined code would be. For many use cases, however, this cost is negligible and the benefit is the ability to write logic in a linear style with `yield`.

Developer experience: From a user perspective, `generator-rs` provides a set of macros and types to make usage ergonomic, but it's still more verbose than Python. You must wrap your generator logic in a closure that the library provides with a `Scope` handle for yielding. For instance, using the crate looks like:

```
use generator::{Gn, Scope, yield_};

let g = Gn::new_scoped(|mut s: Scope<_, i32>| {
    // ... inside generator context ...
    s.yield_(10);        // yield out the value 10
    let val = s.yield_(20); // yield 20, and `val` receives any sent value on
resume
    println!("Received {}", val);
    done!();
});
```

Iterating or sending values uses methods on the returned generator object (which implements `Iterator` for the yield type). This is certainly workable, but less natural than Python's syntax. You must remember to call the `done!()` macro at the end of the generator to mark completion (if you forget, it will abort on drop or cause undefined behavior). Error handling and borrowing rules also introduce some complexity: for example, the crate disallows yielding non-`'static` references unless you use a special "local" generator type and unsafely assert that the usage is sound [21]. In practice, this means if you want to yield borrowed data, you need to ensure the generator is dropped before the data goes out of scope (hence the *scoped* generator variant). These are limitations that a Rust developer must be cognizant of, whereas Python's garbage-collected, dynamic nature doesn't have an equivalent concern – you can yield references to objects and the Python runtime will keep them alive as needed via reference counting.

Safety considerations: The `generator-rs` crate is implemented with a lot of `unsafe` code, as expected. It had to deal with raw pointers, stack manipulation, and even tricks like marking functions `#[naked]` (meaning no prologue/epilogue) to write custom assembly transitions. Unsurprisingly, getting all of this correct is challenging. In fact, an earlier version of the crate had a bug leading to **uninitialized memory** use in the `Scope`, `yield_`, and `done` APIs, which was reported and assigned CVE-2019-16144 [22]. This was fixed in version 0.6.18, but it highlights how manual management of stacks and registers can introduce subtle memory safety issues. The crate's maintainer has been responsive in expanding platform

support and fixing bugs, but users should understand that using such a library relies on the correctness of a large amount of unsafe code. Thankfully, once it matures, it provides a safe interface – meaning typical usage of `generator-rs` (just calling `Gn::new_scoped` and using the generator) does not require the end-user to write unsafe code themselves.

In summary, `generator-rs` demonstrates that Rust *can* have Python-like generators and coroutines with today's stable language, by dropping down to a low level. It offers a rich set of features (yield, send, yield-from, cancellation, etc.) that make it possible to port many Python generator-based workflows to Rust [13] [23]. The cost is the added complexity of a separate stack and context switching mechanism, plus the need to manage lifetimes a bit more manually (ensuring borrowed data doesn't outlive the generator's suspension point). It's a tour de force of low-level programming in Rust, using assembly to fill a gap in the language's current capabilities.

4. From `generator-rs` to Rust's Future: Improving Generators via Crate Techniques and RFCs

The design and techniques of `generator-rs` provide valuable lessons for both the improvement of the crate itself and potential upstream enhancements in Rust.

Improving the developer experience and performance (within the crate): One area of improvement is **ergonomics**. While `generator-rs` achieves its goal, it could be made more ergonomic with additional helper macros or abstractions. For instance, the requirement to call `done!()` explicitly is easy to forget – perhaps the crate could provide an RAII guard or a wrapper that automatically calls `done!()` when the closure finishes (though this is tricky because the closure might finish via `return` or panic). Another improvement could be providing a syntax extension or proc-macro to allow writing a generator in a style closer to native `gen` syntax. For example, a procedural macro could transform a function with `yield_` calls into the necessary closure setup – so a user could write something like `generator! { |args| { yield_(...); ... } }` and the macro handles `done` and creation. This would hide the boilerplate. As for performance, the crate could take inspiration from **libfringe**, which demonstrated extremely fast context switches. If not already done, `generator-rs` might adopt similar assembly routines or optimizations (e.g. using certain instruction sequences that are faster for saving/restoring registers, or minimizing memory accesses during context switch). An issue in the project's tracker actually discussed using libfringe's context switch approach to speed up `generator-rs`, noting libfringe was 10× faster in some benchmarks. Ensuring that optimizations like avoiding unnecessary memory barriers or pipeline flushes in the assembly could help. However, it's worth noting that the absolute performance is likely already very high and the differences might only matter in extreme cases (like millions of switches per second).

Another possible improvement is to reduce the amount of **unsafe code** the user is exposed to indirectly. For example, the crate had to deal with the unsafety of non-`'static` references. It currently provides a separate `LocalGenerator` for non-static usage, which is a safer API than the earlier approach (which involved an `unsafe { yield_unsafe(...) }` as seen in the docs [24]). Further wrapping and hiding such unsafety (or leveraging newer compiler features) can improve safety. As Rust's own unstable generator feature advances, `generator-rs` might even leverage compiler support in the future instead of raw assembly (though that's speculative; by the time generators are stable, the need for this crate's approach may diminish).

Informing Rust language design (upstream RFCs and proposals): The existence of `generator-rs` proves that there is strong interest in full coroutines (with their own stack) in Rust, not just the limited state-machine generators. Rust's current path is to implement **stackless** generators (because they integrate with traits like `Iterator` and don't require extra runtime support). However, the techniques in `generator-rs` could inspire proposals for **first-class coroutine support** in Rust. For example, one might imagine an RFC for a `std::task::Coroutine` that behaves like a lighter-weight thread – essentially what `generator-rs` provides but as an official API. Such an API could standardize the assembly routines or use built-in compiler intrinsics to manage context switching, potentially making it easier to maintain across architectures (or even leveraging LLVM's support if any for context switching). That said, adding true stackful coroutines to Rust's standard library or language is a large step that the Rust team has historically been cautious about – it overlaps with what OS threads do (but without OS scheduling) and what async tasks do (but with their own stack). The prevailing view has been that stackless coroutines (async/await, generators) cover most use cases without adding the burden of stack management. In fact, design team discussions have noted that many use-cases for resume arguments can be covered by other patterns (like shared mutable state) and thus "all they need to implement are generators, not true coroutines" in the language [9]. This suggests that upstream Rust is likely to focus on completing the generator feature (with possible resume arguments) in a stackless manner, rather than adopting stackful coroutines in core.

However, the techniques from `generator-rs` could still inform *supporting features* in Rust. One example is **stack allocation APIs** – the crate currently uses either the C runtime (for Windows fibers perhaps) or raw OS calls to allocate memory for stacks, plus inline assembly to manipulate stacks. If Rust provided a safer abstraction for stack allocation with guard pages (similar to what libfringe's `OsStack` does) or even an unstable intrinsic for context switch, it would reduce the reliance on external assembly. An RFC could propose adding a module in `std::arch` or `std::sys` for performing a context switch between two stack pointers. This would be highly unsafe, but it would encapsulate the assembly in the standard library where it could be audited and optimized by the compiler team, rather than every coroutine library writing its own assembly. In essence, **upstreaming the mechanism** while leaving the policy to libraries could be a way to improve portability and safety. For instance, an upstream function might look like `pub unsafe fn swap_context(old_ctx: *mut Context, new_ctx: *const Context)`, where `Context` holds saved registers. Libraries like `generator-rs` or future ones could use this instead of writing assembly per target.

Another area is **resume arguments in generators (bidirectional flow)** at the language level. The experience of `generator-rs` shows that having a `.send()` API and yield returning a value is useful and can be done. The Rust language could integrate this by allowing `yield` to be an expression that evaluates to the sent-in value (or a default if none is sent, akin to Python's `None` on `.next()`). This will likely come after basic generators stabilize. The design discussions (RFC 2781 and others) have explored this, noting it would unify generators with the closure traits (`FnOnce`, `FnMut`) by treating the resume argument like function arguments on each resume [25]. Solving this at the compiler level would immediately make Rust generators as powerful as Python's (and even more, given static typing). Insights from `generator-rs` – for example, how it handles the first resume (where Python's generator `.send()` requires a `None` for the initial start) – could guide the ergonomics in Rust. The crate's API uses a `raw_send(None)` or simply calling `next()` to start the generator [26], which is similar to Python needing to call `gen.send(None)` or `next(gen)` before you can send real values. Rust might abstract that detail (perhaps by not requiring a separate call to start, if the first yield could take an argument that's provided at creation or defaulted).

"yield from" and composition: The crate's support for `yield_from` demonstrates a desire to easily compose generators. In Rust, if you have an iterator (or another generator), you can already loop over it and yield each item, but it's verbose to do manually. The crate simply provides a convenience. Upstream, there might not be a need for a special language construct (Python needed `yield from` to delegate send/throw as well, which is less relevant if Rust doesn't have throw-into-generator). But it's something library authors can easily add (for example, a macro or method to do delegation). If Rust ever added built-in `yield from` it would likely be sugar for `for item in iterator { yield item; }`.

Coroutine scheduling and multi-threading: Interestingly, the `generator-rs` README mentions a "coroutine framework running on multi thread" as a possibility [23]. Indeed, once you have stackful coroutines, you can build an executor to schedule them across threads or round-robin on one thread. This starts to resemble Go's goroutines or fibers. While that's outside Rust's std library at the moment, it's feasible as an external library built atop `generator-rs`. If such patterns gain popularity, it could influence proposals for a more integrated runtime or officially supported executors. Rust's stance so far has been to leave scheduling policies to libraries (as seen with the async ecosystem where multiple executors exist). The techniques in `generator-rs` could thus inform higher-level library designs (for example, building a channel that works with stackful tasks, or an async scheduler that migrates stackful tasks between OS threads). Upstream contributions might include better hooks for custom executors or improvements to `std::thread` to make user-space context switches easier to manage (though `std::thread` uses OS threads, which are heavier).

In summary, `generator-rs` serves as a proving ground for what fully-featured coroutines in Rust might look like. Its successes and challenges provide guidance to the Rust language designers: **successes** like achieving Pythonic generator functionality (confirming that the concept is sound and useful in Rust), and **challenges** like managing borrowed data and ensuring safety (highlighting what the language needs to solve if it were to natively support these patterns). As Rust's own generator feature matures, we may see some of the crate's capabilities (e.g. resume arguments) absorbed into the official design. Conversely, if there remain use cases for stackful coroutines (such as long-running computations that benefit from separate stacks, or easier FFI integration by having a reentrant call stack), the community might push for an RFC to support them more directly, potentially borrowing ideas and even code (with permission) from `generator-rs`. At the very least, the crate will continue to be a crucial tool for Rustaceans who need Python-like generator behavior right now, and its development experience will be valuable in informing any future Rust RFCs around generators and coroutines.

5. Portability and Platform Neutrality Considerations

One important aspect when comparing Rust and Python in this context is portability. **Python's generators and coroutines are inherently portable** – the logic is at a high level in the interpreter/VM, so any platform that runs Python supports its generators identically. Rust's *language-level* generators (the upcoming `gen` feature) are also platform-neutral, since they compile down to standard Rust code (state machines) that run on any target without special runtime support. However, when implementing coroutines via lower-level tricks (as `generator-rs` does), we must consider platform-specific details.

The `generator-rs` crate, as discussed, includes a lot of architecture-specific assembly. This means its correctness and performance may vary per platform. The maintainers have done an admirable job covering many architectures [18] [19], but if you need to run on an unsupported architecture (imagine a niche CPU or

WebAssembly), you're out of luck unless you implement the assembly for it. Porting to a new architecture requires understanding of that architecture's ABI (which registers are callee-saved, how stack frames work, etc.) and writing assembly to save/restore those registers. For instance, adding PowerPC64 support needed knowledge of the OpenPOWER ABI (as seen in community discussions) to correctly save registers like LR (link register) and CR (condition register) [27] [28] . This is non-trivial and can be a maintenance burden.

In contrast, Rust's native generator feature (when stable) will be implemented by the compiler in a platform-neutral way – it transforms code to a state machine using only normal Rust control flow under the hood. So there's no custom assembly per target; the usual code generation takes care of it. This gives it an edge in portability: a `gen` function will work on any target that Rust supports (from bare metal to WebAssembly) without special considerations.

Portability of the concept of "send" is also worth noting. Python's send is part of the language spec and works everywhere. In Rust, the `Generator` trait's resume argument is also portable in concept, but if you rely on a library like `generator-rs` for send support, you inherit its platform limitations. That said, alternative solutions like genawaiter (built on async/await) are portable to any platform that supports Rust's async (which is effectively all tier-1 and tier-2 targets, including WASM). Genawaiter sacrifices some performance (by using a Future and a waker under the hood) in exchange for *platform neutrality*. This highlights a general theme: high-level, *stackless* coroutine implementations favor portability, while low-level, *stackful* implementations favor performance and flexibility but at the cost of per-platform adaptation.

Platform-specific optimizations: If writing or enhancing a system like `generator-rs`, one should consider platform differences. For example, on x86_64, context switching can use certain instruction sequences (like `mov` to swap stack pointers and `ret` to jump) which are extremely fast, whereas on a RISC architecture you may need a different approach. Some platforms have special instructions or OS-provided functions for coroutines (Windows has fibers, which are user-mode schedulable contexts; using those might be an option instead of custom asm on Windows). On Unix, one could use `swapcontext` or `getcontext` from libc (though they are deprecated on Linux and not always available). `generator-rs` opts for inline assembly likely because it offers more control and less overhead than those C library functions, but a contributor might evaluate if using compiler-builtins or intrinsics could improve portability (for example, leveraging LLVM's coroutine support, if any, though LLVM's coroutines are tied to C++ `std::experimental::coroutine` and not exposed in Rust).

Another portability concern is **calling convention**. The assembly in `generator-rs` must ensure that it adheres to the platform's calling convention to not upset Rust's own expectations. This includes aligning the stack pointer, preserving certain registers, and so on. Bugs in this area can be very subtle. By pushing some of this logic into the compiler (through an RFC, perhaps providing an attribute to mark a function as a "coroutine entry" that the compiler knows how to handle), we could make it safer. So an upstream idea could be an attribute like `#[coroutine] fn f(...)` that ensures the function is emitted in a way that is compatible with manual context switching (for instance, not clobbering callee-saved registers because the swap code will handle them). This is speculative, but it's an example of how recognizing the use-case at the language level could improve cross-platform correctness.

Finally, it's important to consider **future architectures and environments**. As Rust moves into domains like embedded and kernel programming, having a portable way to yield/cooperate could be valuable. Stackless generators will likely be the go-to in constrained environments (since they don't require extra stacks), but if stackful coroutines were desired (say for an embedded scheduler), one would need to implement assembly

for those CPUs (ARM Cortex-M, etc.). This is doable but increases the surface for errors. A platform-neutral design, conversely, could be something like leveraging the existing stack of the thread: for example, an alternative coroutine approach is to use segmented stacks or reusing a single stack with checkpointing – but Rust formally abandoned segmented stacks early on for performance reasons.

In conclusion, the platform-neutral approach (Rust's built-in generators and async) is the most portable and maintenance-friendly, whereas the stackful approach (`generator-rs`) is more performant and flexible at runtime but requires per-architecture support. Both have their place. A savvy Rust developer or RFC author will consider portability by perhaps using hybrid approaches: use high-level generators when possible, and only drop to assembly for the few cases that truly need it, and even then isolate the assembly behind well-defined interfaces. If contributions to upstream Rust were to be made, ensuring any new coroutine feature can work on *all tier-1 targets* (and ideally tier-2) is a must – this was clearly on the mind of `generator-rs`'s author given the breadth of arch support they added. As Rust grows, we may see the gap between these approaches lessen: either through more powerful optimizations for stackless generators or through safer abstractions for stackful ones.

Sources:

1. Rust RFC 3513 (Gen Blocks) – *"reserves the* `gen` *keyword in the Rust 2024 edition…* `gen` *blocks produce values that can be iterated over with* `for` *"* [1]. Example usage of `gen` and motivation to simplify iterator creation [3] [4].

2. Stack Overflow – demonstrating Rust 2024 nightly usage of `gen` functions and blocks with yields [2].

3. RFC 2781 (resume arguments) – *"Generator resume arguments are a sought after feature… absence … forced async-await to use thread-local storage"* [7].

4. Rust language design notes on coroutines – commentary that some languages pass resume args by mutable shared state, implying focusing on generators (one-way) can cover many needs [9].

5. `generator-rs` README – usage example and feature goals (send/yield, cancel, yield_from, etc.), plus supported platforms list [14] [13] [18] [19].

6. `generator-rs` docs – example of using `.send()` in a loop and yielding values [10].

7. RaptorCS forum (PowerPC port discussion) – explanation of `generator-rs` assembly functions (`bootstrap_green_task`, `swap_registers`) and their role in context switching [17].

8. Libfringe README – performance claim of ~3ns context switches on x86_64 [20], illustrating potential optimization.

9. CVE-2019-16144 – note on an uninitialized memory bug in `generator-rs` <0.6.18, highlighting the complexity of implementing such a library safely [22].

10. Rust user forum (announcing genawaiter) – points out genawaiter uses no assembly, only async/await (for portability) [11].

---

[1] [3] [4] [5] [6] 3513-gen-blocks - The Rust RFC Book
https://rust-lang.github.io/rfcs/3513-gen-blocks.html

[2] Lazy sequence generation in Rust - Stack Overflow
https://stackoverflow.com/questions/16421033/lazy-sequence-generation-in-rust

[7] [25] Unified coroutines a.k.a. Generator resume arguments by semtexzv · Pull Request #2781 · rust-lang/rfcs · GitHub
https://github.com/rust-lang/rfcs/pull/2781

[8] [9] Generalizing coroutines - The Rust Language Design Team
https://lang-team.rust-lang.org/design_notes/general_coroutines.html

[10] [15] [21] [24] [26] Gn in generator - Rust
https://docs.rs/generator/latest/generator/struct.Gn.html

[11] Announcing genawaiter – use generators (yield) on stable Rust - announcements - The Rust Programming Language Forum
https://users.rust-lang.org/t/announcing-genawaiter-use-generators-yield-on-stable-rust/34283

[12] [13] [14] [18] [19] [23] GitHub - Xudong-Huang/generator-rs: rust stackful generator library
https://github.com/Xudong-Huang/generator-rs

[16] gen_impl.rs - source
https://docs.rs/generator/latest/src/generator/gen_impl.rs.html

[17] [27] [28] Show Posts - tle
https://forums.raptorcs.com/index.php?action=profile;u=140;area=showposts;start=30

[20] mirrors_steveklabnik/libfringe
https://gitee.com/mirrors_steveklabnik/libfringe?skip_mobile=true

[22] CVE-2019-16144 Detail - NVD
https://nvd.nist.gov/vuln/detail/CVE-2019-16144