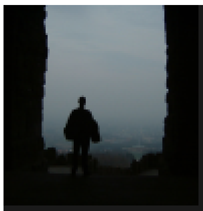

The Rust I Wanted Had No Future

Jun. 4th, 2023 06:11 pm

graydon2

In a recent podcast about Rust leadership, the BDFL question came up *again* and Jeremy Soller said (in the understatement of the century) that "I believe Graydon would have said no to some things we all like now". And this echoes a different conversation on reddit where I was reminded that I meant to write down at some point how "I would have done it all differently" (and that this would probably have been extremely unsatisfying to everyone involved, and it never would have gone anywhere).

Boy Howdy would I ever. This is maybe not clear enough, and it might make the question of whether the project "really should have had a BDFL" a little sharper to know this: the Rust We Got is many, many miles away from The Rust I Wanted. I mean, don't get me wrong: like the result. It's great. I'm thrilled to have a viable C++ alternative, especially one people are starting to consider a norm, a reasonable choice for day-to-day use. I use it and am very happy to use it in preference to C++. But!

There are so, so many diferences from what I would have done, if I'd been "in charge" the whole time.

Agreeable improvements

I'm not just talking about stuff I didn't understand how to do right or didn't quite appreciate the right way to do. There's lots of that. The design was at best half-baked when I went public with it and new people started showing up, and lots of stuff I didn't understand, or guessed the wrong setting for, or didn't have totally working. Sure. Some examples from that list are easy targets for the "more minds make for better designs" argument:

- Move semantics were opt-in rather than the default.
- There was an effect system that mostly didn't work (no effect-polymorphism).
- There was a first class module system that mostly didn't work (lots of fussy technical issues).
- There was a typestate system that turns out to be redundant once you have affine types, nominal types and phantom type parameters.
- The only concurrency was Erlang-style unstructured actors, with no direct parallelism like threads or locks.

Arguably most of this stuff, given sufficient input from others, could have got fixed. Not 100% clear about actors -- I was weirdly focused on that model that in practice has many issues -- but otherwise I could certainly have used help with the design of some of these and given some help would probably have nudged them in a direction we can all agree was better.

Disagreeable divergences

But for a ton of other issues, we didn't agree with what constituted "better", and I still don't entirely agree with what got chosen. This happened I think 4 main ways:

- I sketched-out X (or merely planned to), someone else argued for Y with passion, persistence or social influence to the point that they got their way, and I just lost.
- I sketched-out X, someone else preferred Y and sketched it out, their prototype Y was compelling enough to take the place of X in order to evaluate it, and we were pressed for time and never revisited it to deal with all the shortcomings of Y.
- I sketched-out X, circumstances (usually porting-to-LLVM *shakes fist*) demanded that we do things more like Y, we temporarily put it in place, and again ran out of time or options to revisit so stuck with Y.
- I sketched-out X, it was clearly not-quite-right enough that we ripped it out to avoid people committing to building on bad foundations, this left a vacuum and time ran out and the language officially shipped nothing, and later the ecosystem filled it in (possibly with multiple competitors).

Here are just a few of the places I literally didn't want, and/or currently don't like, what Rust wound up with:

- **Cross-crate inlining and monomorphization**. I wanted crates to allow inlining inside but present stable entrypoints to the outside. Swift wound up close to here, it's a huge technical headache but failure to do so is also a big part of Rust's terrible compile times and lack of a stable ABI. I resisted this at the time and have objected to the choice ever since. It's likely necessary in today's Rust given the next point though.
- **Library-defined containers, iteration and smart pointers**. Containers, iteration and indirect-access operations (along with arithmetic) comprise the inner loops of most programs and so optimizing their performance is fairly paramount; if you make them all do slow dispatch through user code the language will never go fast. If user code is involved at all, then, it has to be inlined aggressively. The other option, which I wanted, was for these to be compiler builtins open-coded at the sites of use, rather than library-provided. No "user code" at all (not even stdlib). This is how they were originally in Rust: vec and str operations were all emitted by special-case code in the compiler. There's a huge argument here over costs to language complexity and expressivity and it's too much to relitigate here, but .. I lost this one and I still mostly disagree with the outcome.
- **Exterior iteration**. Iteration used to be by stack / non-escaping coroutines, which we also called "interior" iteration, as opposed to "exterior" iteration by pointer-like things that live in variables you advance. Such coroutines are now

finally supported by LLVM (they weren't at the time) and are actually a fairly old and reliable mechanism for a linking-friendly, not-having-to-inline-tons-of-library-code abstraction for iteration. They're in, like, BLISS and Modula-2 and such. Really normal thing to have, early Rust had them, and they got ripped out for a bunch of reasons that, again, mostly just form "an argument I lost" rather than anything I disagree with today. I wish Rust still had them. Maybe someday it will!

- **Async/await**. I wanted a standard green-thread runtime with growable stacks -- essentially just "coroutines that escape, when you need them too". Possibly with a somewhat-embeddable outer event loop / IO manager library, but that's always going to be a little tricky. Go started and stayed here, but they had to do a bunch of gory compromises to make the FFI work and it leaves a lot of performance on the table and torpedoes a bunch of embedding opportunities. Rust started here too, and it got rewritten a couple times and eventually thrown out because of a lot of reasons but none which completely obviated the need (as evidenced by the regrowth of Async/Await and Tokio). I've softened my position on this and have a grudging respect for where Rust wound up (especially in enabling heterogeneous selects, which I think puts it in a similar and enviable expressivity class as Concurrent ML). But if I'm being honest I never would have agreed to go in this direction "if I was BDFL" -- I never would have imagined it could even *work* -- and still don't know that I think the result quite pays for itself.

- **First-class &**. I wanted & to be a "second-class" parameter-passing mode, not a first-class type, and I still think this is the sweet spot for the feature. In other words I didn't think you should be able to return & from a function or put it in a structure. I think the cognitive load doesn't cover the benefits. Especially not when it grows to the next part.

- **Explicit lifetimes**. The second-class & types in early Rust were analyzed for aliasing relationships to ensure single-writer / multi-reader (as today's borrows are) but the analysis was based on type and path disjointness and (if necessary) a user-provided address-comparison disambiguation. It did not reason about lifetime compatibility nor represent lifetimes as variables, and I objected to that feature, and still think it doesn't really pay for itself. They were supposed to all be inferred, and they're not, and "if I were BDFL" I probably would have aborted the experiment once it was obvious they are not in fact all inferred. (Note this is interconnected with previous points: the dominant use-cases have to do with things like exterior iterators and library-provided containers).

- **Environment capture**. I often say (provocatively) that "I hate lambda", but lambda is actually (at least) two separate language features: one is a notation for anonymous function literals; another is equipping those anonymous function literals with *environment capture*. I don't really care either way about having such a literal syntax, but I really do dislike equipping it with environment capture and think it's a mistake, especially in a systems language with observable mutation. Early Rust had something called "bind expressions" which I copied from Sather and which I think are better (clunkier but better). Rust gained lambda-with-environment-capture in a single package which I didn't object to strongly enough, as part of trying to make interior iteration work on LLVM-with-no-coroutines, and this motivation was later removed when we moved to exterior iteration. But "if I were BDFL" I would *probably* roll back the environment capture part (it's easier to tolerate for non-escaping closures which many are, eg. in non-escaping coroutines / stack iterator bodies, but .. eh .. complex topic).

- **Traits**. I generally don't like traits / typeclasses. To my eyes they're too type-directed, too global, too much like programming and debugging a distributed set of invisible inference rules, and produce too much coupling between libraries in terms of what is or isn't allowed to maintain coherence. I wanted (and got part way into building) a first class module system in the ML tradition. Many team members objected because these systems are more verbose, often painfully so, and I lost the argument. But I still don't like the result, and I would probably have backed the experiment out "if I'd been BDFL".

- **Underpowered existentials**. The `dyn Trait` mechanism in Rust allows runtime and heterogeneous polymorphism, a.k.a. Existentials. These types are useful for many reasons: both solving heterogeneous representation cases and also selectively backing-off from monomorphization or inlining (when you have those) in order to favour code size / compile time or allow dynamic linking or runtime extension. Early Rust tried to use these extensively (see also "first-class modules") and actually had an intermediate-rigidity type called an `obj` that was always a sort of Cecil-like runtime-extensible existential glued to a self-type record that allowed method-by-method overriding at runtime (almost like a prototype-OO system). Today's Rust strongly discourages the use of any such dynamic dispatch, a feedback loop arising from both technical limitations placed on them and a library ecosystem that's taken that as a sign never to use them.

- **Complex inference**. Long ago Rust's website was just a feature list, and it had two entries that read "Type inference: yes, only local variables" and "Generic types: yes, only simple, non-turing-complete substitution". The first should actually have read "statement-at-a-time" but obviously that wouldn't have held up any more than the rest of it. I don't like "playing type tetris" and I would prefer it went back to this position, though probably the substitution would have needed to at least support bounds on module-typed parameters. I think there's a design space that avoids having to make the user think about unification or at least recursion in their types. Anyway we got a lot of type-system people who think complex types are a good thing, or at least an excusable one, and I completely lost this argument.

- **Nominal types**. Another item on that list claims that the type system is "structural". In general I think structural types are better, and we used them a lot in early Rust. There are reasons for a few special nominal sites in a structural system (eg. modeling typestate) but in general I think Rust went too far into the Nominal camp (in part to support traits) and this has damaged its options in compositionality and polymorphism.

- **Missing quasiquotes**. The language was intended to ship with something like quote and we just didn't get there in time. The whole static metaprogramming system really shipped in a half-done state.

- **Missing reflection**. Similar to the model proposed in "a mirror for Rust", the language initially had (and I hoped it would have again) complier-emitted "type descriptors" that the user could invoke a reflection operator on. Once compile-time execution got good enough this could be cheap; when we had it it involved emitting runtime code and that codegen was actually why we took it out: too much time spent in LLVM generating glue code nobody was using most of the time. But that's a bad reason to omit it as a general feature!

- **Missing errors**. The error-handling system that shipped in 1.0 was basically a void resulting from ripping out a couple previous attempts, and it only got filled in later (the 2018 edition) with a copy of the ? operator from Swift. This is only a half-copy though; the `throws` syntax, the dedicated error-carrying ABI, and the not-necessarily-allocating existential Error type in Swift would have all helped to fill in this "explicit error union" model. And "if I were BDFL" I might not even have wanted to go there, I had a few other feelings about errors. Anyway the result isn't what I wanted at any point, and I don't know where I would have gone with it.

- **Missing auto-bignum**. Another thing that's great to have a compiler open-code is an integer type that overflows to an owned or refcounted bignum type: shipping enough stuff to let this happen efficiently in libraries is a huge pain (even if you get as far as stable inline assembly it won't go as fast as doing it in the compiler) and .. Rust just decided not to. I wanted it to, but I lost. Integers overflow and either trap or wrap. Great. Maybe in another decade we can collectively decide this is also an important enough class of errors to catch? (Swift at least traps in release by default -- I wish Rust had chosen to).

- **Missing decimal FP**. More minor but basically every language discovers the long way that financial math is special and, at great length, eventually adds a decimal type. I wanted Rust to do this up front, but it was perpetually deferred to libraries. There are a few, but it'd be better to have one built in (so you can have literals and interop and so forth).

- **Complex grammar**. I've become somewhat infamous about wanting to keep the language LL(1) but the fact is that today one can't parse Rust very easily, much less pretty-print (thus auto-format) it, and this is an actual (and fairly frequent) source of problems. It's easier to work with than C++, but that's fairly faint praise. I lost almost every argument about this, from the angle brackets for type parameters to the pattern-binding ambiguity to the semicolon and brace rules to ... ugh I don't even want to get into it. The grammar is not what I wanted. Sorry.

- **Tail calls**. I actually wanted them! I think they're great. And I got argued into not having them because the project in general got argued into the position of "compete to win with C++ on performance" and so I wound up writing a sad post rejecting them which is one of the saddest things ever written on the subject. It remains true with Rust's priorities today, I doubt it'll ever be possible across crates (maybe maybe within), but as with stack iterators IMO they're a great *primitive* to have in a language, in this case for writing simple and composable state machines, and "if I were BDFL" I probably would have pointed the language in a direction that kept them. Early Rust had them, LLVM mostly made us drop them, and C++ performance obsession kept them consigned to WONTFIX bug status.

Themes

Maybe you've nodded your head about one or two of the things above and think they're good ideas -- "hey, maybe we *should* have had a BDFL!" -- though more likely you think they're terrible. But the point of this post isn't just to lob a bunch of suggestions about direction at the current project, or grind a bunch of long-dull axes, or even to make myself look bad in public.

The point is to indicate *thematic divergence*. The priorities I had while working on the language are broadly not the revealed priorities of the community that's developed around the language in the years since, or even that were being-revealed in the years during. **I would have traded performance and expressivity away for simplicity** -- both end-user cognitive load and implementation simplicity in the compiler -- and by doing so I would have taken the language in a direction *broadly* opposed to where a lot of people wanted it to go.

Performance: A lot of people in the Rust community think "zero cost abstraction" is a core promise of the language. I would never have pitched this and still, personally, don't think it's good. It's a C++ idea and one that I think unnecessarily constrains the design space. I think most abstractions come with costs and tradeoffs, and I would have traded lots and lots of small constant performancee costs for simpler or more robust versions of many abstractions. The resulting language would have been slower. It would have stayed in the "compiled PLs with decent memory access patterns" niche of the PL shootout, but probably be at best somewhere in the band of the results holding Ada and Pascal.

Expressivity: Similarly I would have traded away so much expressivity that it would probably make most modern Rust programmers start speaking about the language the way its current critics do: it'd feel clunky and bureaucratic, a nanny-state language that doesn't let users write lots of features in library code at all, doesn't even trust programmers with simple constructs like variable shadowing, environment capture or inline functions. Ask people who worked on the early compiler what it was like. It was not easy or fun, and I was stubborn about adding cognitive or implementation complexity just to make it a few percent easier or more fun. I didn't mind making users write the same thing over and over, or write declarations out in full, or write something as multiple separate statements with separate variables instead of one complex nested expression.

Divergence in preferences are real! My preferences are weird. You probably wouldn't have liked them. Look at the design goals in the early versions of the manual or even later versions. If I'd stayed in charge (or even asserted a more robust sense of "being in charge" when I was *nominally* moreso) the result would have been, I think, fairly unpopular. The Rust I Wanted probably had no future, or at least not one anywhere near as good as The Rust We Got. The fact that there was *any* path that achieved the level of success the language has seen so far is frankly miraculous. Don't jinx it by imagining I would have done any better!