## Learn WebAssembly …by writing small programs

(github.com/emnudge)

468 points by todsacerdoti 7 months ago | 81 comments

***ReleaseCandidat*** Because I've read it here and elsewhere quite often: What is missing from WASM in the browser isn't "just" DOM access, but "everything" else - including fetch or XMLHttpRequest - too. Here is a list of all web APIs supported by browsers and not supported by WASM, the DOM is just one of them: https://developer.mozilla.org/en-US/docs/Web/API

But at least there is hope, that these interfaces will be available once GC is final and supported by browsers:

Once GC is supported, WebAssembly code would be able to reference and access JavaScript, DOM, and general WebIDL-defined objects.

Last paragraph at https://webassembly.org/docs/web/***andsoitis*** GC proposal is from 2018: https://github.com/WebAssembly/proposals/issues/16 and there’s code: https://github.com/WebAssembly/gc/blob/master/proposals/gc/O...

Seems like an awefully long time for progress to be made, given all the possibilities it would unlock.

***ReleaseCandidat*** Yes, but it is now (for some months) supported by V8 (browser, Node and Deno - hidden behind a flag) and Wasmedge and Wasmtime will both have implementations soon.

But I wouldn't hold my breath to being able to call web APIs directly from WASM ;)

***tabtab*** If they "solve" that then they risk being the next Java Applets.***cedws*** Funny how have things have progressed. We went from using HTML + CSS for simple presentation of information, to enriching with small amounts of client side scripting, to plug-and-play sandboxed applications with Flash/Java. Then we realised those were a terrible idea, Flash and Java Applets died, so we started treating JavaScript as a compilation target to mount behemoth abstractions enabling a native application-like experience. When browsers collapsed under the weight of all that crap, WASM was born by necessity. And so we're back to applets.

Am I getting this right?

***tabtab*** Our current web standards are ill-suited for what we really want out of the web (rich desktop-like GUI's), so people keep trying different things in desperation. One thing not really tried yet is a state-ful GUI markup language standard. Also, maybe applets could be done right on a second try, applying lessons of the first.***Hadriel*** I haven't been following webassembly, but as the former 'cure' for JS, why has it taken so long to get the basics?***ruune*** It's niche. Front end devs want to use JS or equivalents, not back end languages. And back end devs don't want to browser stuff. So only full stack devs left***ysavir*** Looks very similar to the Exercism model, which also has a free WASM course filled with small exercises[1]. I wonder if the author considered contributing to that course or working together with them, it might get their work to a broader audience and leverage the existing toolset Exercism has to offer.

[1]https://exercism.org/tracks/wasm

***emnudge*** I really like Exercism! Unfortunately their exercise model is considerably more free-form where it teaches you considerably less and in much larger chunks. I think this is a great model for certain contexts, but the format I have in the repo is more similar to "rustlings" and "ziglings" where you're taught syntax and features alongside the code examples.

I don't know that their Wasm module is necessarily "broken", so I'm unsure whether my contributions would be welcome.

***ysavir*** Very fair! It's been close to a decade since I used Exercism so I don't really know what their stuff is like nowadays. From what I remember, the focus was less on teaching you the language directly (via instructions) and was more on encouraging the user to reiterate on their code and experiment, and letting reviewers lead the direction of growth.***raincole*** One thing I don't like about Exercism is that except for the most popular languages, the exercises are often not "sysematics".

In other words, it's just a bunch of leetcode-like questions, ordered by difficulties.

A proper course should order the exercises by language features. Exercism actually has built a fantastic interface for this[1], but not utiltized it for most langauges.

[1]: https://exercism.org/tracks/csharp/concepts

***tietjens*** I completely agree with this. I love the platform, the mentors, and the enthusiasm for languages. They’ve done a great series every month this year on a new set of languages, and have great YouTube interviews.

However, the problems they offer do not help you learn the syntax or quirks of specific languages and for me are only a place to practice leetcode/codewars style logic. Also important! But not what I need when I first begin a language.

***rofrol*** But leetcode has this https://leetcode.com/studyplan/***imiric*** This is great, thanks!

One of my favorite ways to learn a new language or framework is via "koans"[1], which this reminds me of. It's a gentle ramp up from basic to advanced features, and the TDD-like workflow of seeing a test fail, understanding why, and fixing it, is really conducive to learning, while giving you that dopamine jolt from "a-ha!".

[1]: https://github.com/ahmdrefat/awesome-koans/blob/master/koans...

***dfee*** sounds like a great way to learn a language "on your own". unfortunately, Rust is missing. can anyone propose a link?***billti*** It credits Rustlings at the end of the page linked to (https://github.com/EmNudge/watlings#credits).***Kalq*** Isn't that basically Rustlings?***dfee*** i think you've found what i'm looking for! https://github.com/rust-lang/rustlings***ReleaseCandidat*** Just a remark: if you want to play with WASM features like GC, use Binaryen's `wasm-opt` instead of WABT, as `wasm-opt` supports way more WASM extensions.***arek\_nawo*** Pretty cool.

I haven't really explored WASM hands-on (I'll give this guide a try) but, given that it's already been a few years, I think it's been hugely beneficial for web development.

Not the "JavaScript killer" some where hoping for, though it was never meant to be one. Instead it integrates pretty nicely within the existing ecosystem, optimizing existing use-cases and allowing new ones when heavy computations are required. Net benefit for all web devs - faster libraries, impressive dev tools and more portable node binaries.

***danielvaughn*** I've been watching WASM from afar, and it's my understanding that it's not a JS killer because it doesn't have direct access to the DOM or to most DOM APIs. I'm curious if there's another reason I'm missing?***math\_dandy*** Is being a "JS killer" even a goal of WASM? From what I understand, WASM's goal is to allow computationally heavy workloads (e.g., image and audio processing) to the browser by providing a compilation target for C code. This is how we get nice things like Figma or MS Office running in the browser.

Let WASM do the number crunching, let JS do the UI.

***Philip-J-Fry*** It's not the goal from what it seems. But it's definitely what would make it the most interesting to a lot of people, and it's what I think most people expected at some point.

For a lot of companies the only place JavaScript is ever used is on their website frontend. And it makes you wonder, why does it even still need to be JavaScript? With the rise of SPAs and the fall of normal document based websites, browsers are basically just becoming their own platform like Windows Desktop, Mac, Linux, Android, iOS, etc. You could say it's been that way for a long time, but more and more apps are becoming web based only because the browser is now powerful enough to run what used to be a desktop application.

Browsers are literally just a VM with an address bar. We go to a URL and run a program, except right now there's basically the limitation that the program has to be written at least partly in JavaScript. Being able to deploy an entire website as WASM is just the next logical step from what I see.

***math\_dandy*** The two language problem is, indeed, awkward.

DOM bindings in WASM would be awesome. I would be a happy nerd if I could do all of my web dev in OCaml.

In the meantime, WASM has real, just not for front end dev.

The two language problem is hardly unique to web dev. It’s also ubiquitous in the machine learning/data science space with Python and C/C++/CUDA playing the roles of JS and WASM, respectively.

***ReleaseCandidat*** > DOM bindings in WASM would be awesome. I would be a happy nerd if I could do all of my web dev in OCaml.

DOM is not enough for that. You almost certainly would like to be able to communicate with your backend ;)

Here is a list of the "usual" web APIs: https://developer.mozilla.org/en-US/docs/Web/API And everything that needs network access or access to local resources (file system in the worst case) will never happen to WASM because of security considerations.

***xigoi*** How is allowing WebAssembly to use the network any less secure than allowing JavaScript to use the network?***lxgr*** The issue here is the same-origin principle, which rules out most low-level networking (since you could just bring your own SOP-ignoring HTTP client).

Personally I think that enforcing the SOP at all cost (and even when no cookies or other authentication headers are injected by the browser) is misguided at this point and holding back modern webapps.

***ReleaseCandidat*** Don't ask me :).

At least WASM will get DOM access (and hopefully access to similar web APIs) as soon as the GC is stable and usable.

Once GC is supported, WebAssembly code would be able to reference and access JavaScript, DOM, and general WebIDL-defined objects.

https://webassembly.org/docs/web/***sebastianz*** But you will be able to do network access though the fetch API, and fs access though its own API.***catlover76*** > I would be a happy nerd if I could do all of my web dev in OCaml.

You don't like Bonsai? Or do you just want something that doesn't "transpile" to JS at all?

***math\_dandy*** I find that these compile-to-JS languages are great until you want to bring in other JS libraries. Then you’re stuck writing bindings which is annoying, and often brittle if you’re not really careful.***catlover76*** Oh I didn't even think of that--that sounds like a huge pain in the ass lol***zarzavat*** You can certainly bridge the DOM to WASM with a virtual DOM. The problems with WASM are not just interoperability.

The biggest problem is tooling. You cannot build tooling for in-browser WASM because it runs in a browser sandbox. JS has the same problem but the difference is that JS has a known object model that the browser can provide good tooling for.

Whereas with WASM the browser has little insight into what those opaque Memory objects contain. So you need to bring your own tooling and run it inside the sandbox, and the sandboxing does not make this easy.

Let’s say for example you want to pause execution, inspect the object tree, and run a REPL while it’s paused.

***yieldcrv*** If it was the JavaScript killer it will attract JavaScript devs and they’ll make it just like JavaScript

not syntax-wise just community wise, where the package and package managers are a mess and why there is demand for a JavaScript killer

***lenkite*** Yep. Give access to the DOM - perhaps via a batch update/fragment mechanism and watch as it becomes a JS killer. But WASM is currently a starved prisoner locked up in a room only allowed to view the goodies outside the cell door.***tcfhgj*** What would that change? DOM interaction can and is abstracted away by frameworks anyways and not performance critical***ReleaseCandidat*** It doesn't have access to \_anything\_ (in the browser, in other runtimes there is WASI with POSIX functions). Everything has to be imported from or exported to JS.

And currently using anything but C, C++ or Rust isn't feasible, as the runtime needed for a GC is way too big. A Haskell "Hello World" for example is about 1MB (even after running `wasm-opt` on the generated WASM binary).

***progmetaldev*** I do agree things like the GC are very large if we're comparing against standard JavaScript, so I don't believe the technology is ready for standard customer facing websites. For things like internal applications/extranet type applications, I think that the runtime download cost is minimal compared to the functionality that you are given with a proper framework (I can only speak of Microsoft Blazor, but that's just language ignorance, and I know there are other that fit the model as well). As a web developer that also writes utilities to run on the desktop, for things such as ETL or fixing bad data, being able to not switch to another language or even really framework is a huge boon for my time. I know JavaScript, but being able to rarely have to deal with it keeps my head in what I'm solving, rather than having to context switch between interface and server.***yebyen*** > isn't feasible

This is kind of subjective, and in the context of "is this a JS killer" which is what you're answering, I'd agree, it makes sending the Wasm to the browser a bit of a non-starter that it requires a large bundle most of which is simply boilerplate for whatever runtime, without some type of local storage and persistent cache it's difficult to imagine using Ruby in a Wasm for example. If you're deploying to a container, where you're able to use a cache warmer to ensure the wasm is ready before it's called, then a 1mb binary might not be such a big issue.

(I mean, it is still a big issue because the point was fast cold starts, and big assemblies mean no fast cold starts, but again, subjective value judgments... 1mb isn't too big in many cases and I'd wager most of the cases that I'd really care about. But in a browser...)

But if you're not trying to run the Wasm in a browser then it's still potentially quite useful. You might not be running all of your Wasm in a browser and it might still be a JS killer, and all of those might not be in conflict.

***ReleaseCandidat*** Well, the main reason why I'm taking a deeper look at WASM is because I'm creating a language and compiler that is optimized to compile to WASM. While I don't think that my compiler will be the JS killer, I do think that WASM languages using a (the WASM) GC have a future.***patmorgan23*** I mean Microsofts Blazor framework can use WASM to run C#(and its runtime) in the browser.***titzer*** Virgil compiles to Wasm and brings its own GC as well, and the runtime is on the order of about 4KB. The GC is a simple semi-space copying collector and the compiler prunes away dead code from libraries, so binaries are pretty small. So overall I don't think this is a Wasm issue as it is mostly a language runtime size issue.***plopz*** In the browser, can it integrate with SharedArrayBuffer, worker's postMessage or transferrable offscreen canvas?***ReleaseCandidat*** Depends what you mean by "integrate". It always has to import or export JS functions or memory. WASM (without WASI) has no direct connection to "the outside World", everything has to be routed via JS FFI. So you can import or export a SharedArrayBuffer to communicate with JS. https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScr...

But you need the WASM thread extension (for atomic access to the shared memory) to be able to use shared memory.

***connicpu*** WASM essentially can call javascript functions that were imported, and I believe javascript is able to read WASM's memory (a big help for transferring strings). If you're using something like Rust, all the glue code to call any JS APIs can be automatically generated for you with wasm-bindgen, so it really isn't a huge problem usability wise. It's just not great for performance.***dragonwriter*** > And currently using anything but C, C++ or Rust isn't feasible

Someone should tell Anaconda that they can't do this, then: https://pyscript.net/

***tyingq*** That uses Pyodide. From Pyodide's own pages:

"At present a first load of Pyodide requires a 6.4 MB download, and the environment initialization takes 4 to 5 seconds."

So, yes, it works. But there's plenty of situations where a big download followed by a 5 second stall is a non-starter.

***RagnarD*** I'm cheered by ongoing progress in adoption of WebAssembly. While Microsoft is often given minimal notice here, I urge anyone interested in WASM to experiment with Blazor WebAssembly. It's an incredibly powerful framework that lets you use C# and most .NET libraries (including NuGet packages) as compiled WASM in the browser.***hsbauauvhabzb*** Re ‘most nuget packages’ What’s the limitation here? If it requires the ability to read a file on disk, does it straight up fail or does the file read interface push the action back into a server-side rendering type action?***RagnarD*** The NuGet packages are zip files containing compiled binary and possibly other resources. They're referenced in the source code and included at compile time. Since they become integrated into the overall compile, there's no need to reference them in the final runtime product.***hsbauauvhabzb*** Sorry that misses my point, blazor runs client side which means that it’s sandboxed with the same permissions as JavaScript executed within a page - you can’t use blazor to read a file client-side, but there’s nothing to prevent wrapping what would be a server side library in a way where ‘it executes on the client except when it can’t, at which point it executes on the server’

Doing so would almost always be a security risk, as validation would presumably still run client side, but that doesn’t mean it’s not possible to actually achieve such an engine.

***cyanydeez*** Like most wasm, if the nugget is just a thin wrapper around a recompiled binary, it's not going to work.

WASM is a compile target. So if the code source is available and it's "pure", most wasm can now compile it.

Check out pyiodide for a extensive python project trying to bring everything in Python to the web.

Your specific concern regarding file access: they just create something like a indexeddb file system if you want to work with files.

***oefrha*** From reading about it, it seems to suffer the same problem as Go compiled to wasm: payload is huge, usually multi-MB after compression.***cyanydeez*** Is there an article comparing other compile targets.

I mostly get the impression that the compiler in browser is the problem.

If it's adopted natively, that should disappear.

All I'd need is Firefox, edge and chrome to switch to publishing wasm.

***eliben*** Cool project!

I maintain a somewhat related repo here: https://github.com/eliben/wasm-wat-samples/

***emnudge*** That looks super useful! I wish I knew about this when first learning wat. This would make a great reference!***syndicatedjelly*** Gave WASM a try and ran into problems exposing connections to a SQLite database I wanted to connect to locally. Has anyone done something like that before and can point to good resources?***yebyen*** Check out Spin! It includes sqlite support natively in the runtime since 1.4:

https://www.fermyon.com/blog/spin-v14

***syndicatedjelly*** Looks like it’s only available for Python, I was hoping for a more language agnostic solution. Thanks though!***yebyen*** It is? The docs example shows support for Rust, TypeScript, and Python:

https://developer.fermyon.com/spin/sqlite-api-guide#using-sq...

I'm not sure what holds back from being a completely language agnostic solution, but I know when I tried binding functions into my Ruby app with wasmer I had nothing but problems, I wound up using WASI exclusively to communicate with my WASM modules. I don't honestly know how good the WASM language agnostic story is today. I gave a talk with my perspective as a Rubyist at OSS Summit and CDCon/GitOpsCon a few months ago in Vancouver. The talk (and lightning talk variant) are called "Exotic Runtime Targets: Ruby and Wasm on Kubernetes and GitOps Delivery Pipelines"

***brundolf*** It's super interesting to me how much WebAssembly resembles a "real language" that's sorta writable by-hand. I bet it lowers the bar quite a bit when targeting it***frithsun*** Very neat.

As WebAssembly becomes the lingua franca of different ecosystems, having a strong grasp of how it works is worth the time investment. Thanks for this.

***ozcanay*** Cool. Did anyone try targeting Qt applications for WebAssembly? I wonder how convenient it is to use.***zdragnar*** There's an interesting write up here: https://www.linkedin.com/pulse/qt-wasm-nicholas-petreley-1c

Personally, I didn't find the author's sample app to load very quickly- I think it was 10 seconds or so just to get something on the screen- and it was useless on my phone.

That same app would easily perform better and have better device support were it written in js/html/css.

I just don't see a place for things like QT or native gui stuff in real world WASM. Number crunching or fast data processing, sure, or maybe a game, but a standard UI is crap if the whole thing is in a big canvas.

***ducktective*** Is there a WA UI framework for web similar to Svelte or Vue?***RetiredRichard*** Technically Blazor

https://learn.microsoft.com/en-us/aspnet/core/blazor/host-an...

***Kennnan*** Hand written wasm is too low level to use a UI framework with, but there are rust frameworks that compile to wasm for web. For example, Yew and Leptos are both web-first frameworks, and there are a few such as Dioxus or wgpu that are native-first and work on web.***no\_wizard*** What makes hand written WASM too "low level"?

It isn't, really, what the Rust frameworks do is compile down to a specific interop, sure (JS + WASM, its not pure WASM due to lack of DOM access for starters).

That said, no reason a hand written WASM app isn't feasible, with appropriate JS glue

***Joker\_vD*** While it's an interesting (and effective) approach to learn something, what is the point of learning WebAssembly? After all, I thought that WASM was supposed to be something like "LLVM for web", a low-level IR for webdev languages to target and for browsers to execute efficiently, and not something that you'd write manually. Or am I wrong?***ReleaseCandidat*** Actually it is quite high level (with the GC extension even "higher" than C), not comparable to "real" assembler. It has types. It isn't fun writing WAT, but for small stuff it's perfectly doable.

That's a record with 3 fields, its constructor and getters using the GC extension:

;; A `struct` with 3 fields, all immutable.

(type $testStruct

(struct

(field $first i64)

(field $foo f32)

(field $baz f64)))

;; Constructor. `ref` is a reference

(func $mkTestStruct

(param $a i64) (param $b f32) (param $c f64) (result (ref $testStruct))

(struct.new $testStruct (local.get $a) (local.get $b) (local.get $c)))

(export "mkTestStruct" (func $mkTestStruct))

;; Getter function for the three fields:

(func $getTestStruct1 (param $rs (ref $testStruct)) (result i64)

(struct.get $testStruct 0 (local.get $rs)))

(export "getTestStruct1" (func $getTestStruct1))

(func $getTestStruct2 (param $rs (ref $testStruct)) (result f32)

(struct.get $testStruct 1 (local.get $rs)))

(export "getTestStruct2" (func $getTestStruct2))

(func $getTestStruct3 (param $rs (ref $testStruct)) (result f64)

(struct.get $testStruct 2 (local.get $rs)))

(export "getTestStruct3" (func $getTestStruct3))

***emnudge*** Author here!

My initial motivation to learn WASM (as someone from a primarily web background) was that I had a pretty poor understanding of WASM in general and so I had a lot of difficulty working with WASM builds in just about any capacity other than a heavy JS wrapper.

There are aspects to how WASM works that are quite different from other kinds of assembly formats that make learning the basics pretty important. e.g. how memory is requested, provided, grown. How functions are received and exported. Capabilities of tables.

A lot of this might be abstracted by massive wrappers, but you're losing a lot in perf and debugability when using them.

***wanderlust123*** Thanks for putting this together, I too learn best by doing, and this looks very helpful!***chrisco255*** Sometimes you may need to debug the actual wasm code or understand what the compiler is outputting. Depending on the flags you set during compilation, you may want to optimize for size of binary or for performance. You may also want to compare the wasm produced by two different languages or different versions of the language compiler. Maybe you want to verify what libraries are actually getting bundled with your wasm.***thamer*** Understanding what the compiler outputs is also essential to debug the inevitable issues that come up when porting an existing program to WASM. Since it's a different platform with significant limitations, it's important to know how to replace some core APIs and capabilities and how to interface with these alternatives.

By limitations I mean things that a POSIX program would expect that aren't generally available from WASM, like network programming, file system access, processes and threads, pipes and signals. Emscripten and WASI help to some extent, but the replacement APIs are rarely fully compatible.

***hoten*** Some reasons to understand the textual representation, besides curiousity:

You are embedding a WASM engine and want to better grok how the import or exporting works.

You are writing a compiler that targets WASM.

|||\_\_s||| For awhile I was using a raw wasm program for a game engine's random number generator. Stored all the state in global variables, so no heap necessary

Eventually I moved the whole game engine into wasm & at that point it all got ported to Rust

https://github.com/serprex/openEtG/blob/8b7069cc52ec8db3406a...

***mdswanson*** Is it just me, or does "WebAssembly" seem like an oxymoron? Maybe it's because I've been involved with the web since its earliest beginnings.***teddyh*** You might like this bit of speculative history: <https://www.destroyallsoftware.com/talks/the-birth-and-death...>***mdswanson*** Thanks! That was fun. I hadn't seen it. There's the problem of bootstrapping METAL, but perhaps left for a different lecture! ;-)***patmorgan23*** I mean technically it's browser VM bytecode by it's semantics.

## Leaving Haskell behind

321 points by mpereira 7 months ago | 385 comments

***pyrale*** As someone that has also written haskell for about a decade and moved away from it as a breadwinner recently (but for other reasons - I simply wanted to filter job offerings based on social utility rather than language stacks), I definitely agree with the author's first point: the Haskell community values learning extremely strongly. That's great because you work with curious people that have always something to teach and learn. But the community is not so strong when it comes to discard ideas after trying them, and so a professional haskell codebase, if not curated strictly, often ends up with lots of things that you can do but that you probably shouldn't use.

I, however, disagree about tooling. Haskell's tooling sucks, but having used several other languages since (python, js, java, rust, elm), most tooling sucks. After this tour, I miss the Haskell toolchain. Sure, cargo is great, but that's one in many, and most older languages don't have this. I wonder whether Rust can escape this fate as it ages. The author also mentions being dismayed by Python, so I guess that's mostly me seeing the glass half-full though.

***Tainnor*** I've also used several languages and IMHO, Haskell's tooling is some of the worst. Language server breaks with random errors all the time for me, I'm always confused about whether I should have ghcup or stack manage my Haskell versions (and what the benefits and drawbacks are), and so on. I have a project I work on from time to time, and I'm 100% sure that the next time I'll open it up, it will stop working again.

Python's mess of dependency and venv management system is probably equally bad, although my experience was that Poetry is decent.

By contrast, Ruby tooling usually works well (you can choose between chruby, rbenv and rvm, but they all work similarly, and Bundler just works(TM)), and Java has its warts but one of the best IDE experiences I've ever seen. Maybe the refactorings aren't as safe as in Haskell, but they're extremely easy to do.

***js2*** It's weird how personal this is. Sometimes I wonder if it comes down to familiarity.

I've been using Python professionally since 2000 and have never run into issues with its tooling. It's even easier these days: "python -m venv" and then pip are all I need in 99% of use cases. For local development I use direnv + pyenv. I typically develop on macOS and deploy to either macOS or Linux. I previously gave pipenv a try but found it to be brittle and more trouble than it was worth. I haven't used poetry just because I haven't had the need.

Conversely, I regularly run into trouble with ruby. Here's a recent example where I had to contribute a patch to pre-commit to get ruby to install gems where pre-commit wanted it to:

https://github.com/pre-commit/pre-commit/pull/2905

I got so annoyed with bundler I wrote a simple shell script to provide the moral equivalent of Python venvs to gem:

https://gist.github.com/jaysoffian/3c67711d3f00c364365905d87...

All that said, these are just minor annoyances. The only tooling I truly despise is Gradle and especially everything around the Android Gradle Plugin.

***movpasd*** I don't have any issues with Python's packaging ecosystem anymore, having settled comfortably into a pyenv+virtualenv+pip-tools as my "stack" after going around the block a few times.

But even so, I must recognise how awful the experience is for new users. It's taken me years to settle into this system, and it can take half a day to get someone up to speed with these tools if they haven't used them.

I also work a lot with non-developers who need to use or contribute to Python models, so that doesn't help — but I bet it would take an order of magnitude less time to get them up to speed with something like cargo. Coaching them has helped me see how user-hostile the process is to beginners.

It also doesn't help how infectious "all-in-one" Python distributions like Anaconda can be, to the point that whenever anyone has an unexpected issue one of my first reflexes is to check their PATH. The fact that Rust has a widespread default toolchain multiplexer completely solves this issue.

I appreciate that maintaining such a toolchain is work, and there is value as well in the diversity and choice of an open ecosystem. But perhaps building is one place where first-class support by the reference implementation creates a worthwhile tradeoff.

***dcow*** As someone new to using Python professionally after having used it here and there over the course of 15+ years, I’ve run into exactly this problem. It’s pretty standard for a language these days to bundle the dependency manager and build tooling. Python still does this via shell infection. And since there’s 5 different ways to do it it can leave someone trying to figure out what the right vibe is in 2023 spending hours reading about the pros and cons of everything. And all that just to land back on venv+pip+requirements.txt.

Python needs a cargo. Is Poetry it? I’ve been meaning to try it…

***movpasd*** I've tried Poetry, the CLI is nice and user friendly to be sure, but it has many flaws that made me ultimately avoid it.

In my opinion, it tries to be far too "magic", hiding everything about the virtual environment and dependency management within it. This means that if anything breaks (and it does break!), I find it basically impossible to fix it without wiping everything and rebuilding the environment from scratch.

I find the documentation nice and clean to look at but somewhat clunky to navigate and very incomplete or not explicit enough — especially for beginners. For a tool that seems to want to be a "one-stop shop", you need to know a good deal already about Python packaging already to navigate it. The lack of explicitness also makes it hard to troubleshoot (as evoked above).

These are by far my biggest complaints, with the below bring mostly minor gripes.

Additionally, it seems to be targeted at library developers rather than applications (a distinction which to be fair none of the Python ecosystem makes, to the detriment of DX). This doesn't usually end up being a big issue as it's reasonably easy to "librarify" applications, but it's something else to learn. (Not every project wants wheelification as its end goal.)

The usage of pyproject.toml is nice, but the semantics of the file isn't quite as standardised as one might hope, especially when it comes to reflecting the package you're writing in the virtual environment. And the documentation around pyproject.toml is scattered and sparse. (A PEP is not documentation!)

I've complained a lot, so I should round off by saying that Poetry is a commendable project and I have high hopes for it; but as it exists now it's just too rickety for me to recommend it. But others may disagree, and by all means I'd give it a try for your next project.

My personal recommendation remains pyenv + virtualenv (or venv) + pip-tools, with simple projects just using a requirements.in files and then moving to pyproject.toml once it reaches a modest degree of maturity.

***impulser***\_ Have you tried Rye?

https://github.com/mitsuhiko/rye

This is probably the best package manager I used for Python. It feels a lot like Cargo. It sticks to the standards of Python. No custom lock files ect. Uses prebuilt Python so you don't have to build it. Handles global installs easily.

***movpasd*** This is very interesting indeed! A lot of the design choices fix issues I've also personally encountered. The "experimental" dissuades me from using it for real projects, but I'll be keeping an eye on it.***js2*** > Correctly installed, rye will automatically pick up the right Python without manually activating the virtualenv. That is enabled by having ~/.rye/shims at higher priority in your PATH.

I dislike this pattern. I don't want every language/tool manager in my PATH all the time. I much prefer the direnv. I hook only direnv into my shell. Then I can do what I need with an `.envrc` in each project directory.

***dcow*** I hate shell infection because 1) it only works in the shell, 2) it’s not stateless, and 3) depends on your working directory. I mean there’s a reason it’s called “shell infection”. Sounds like Rye specifically aims to address that issue. To each their own, I guess.***systems*** why do sometimes people say things like "and it can take half a day to get someone up to speed with these tools if they haven't used them"

half a day is like almost no time at all

half a day is just few hours, how is this a long time .. how is this any time at all

makes me doubt myself a bit, am i too mediocre to think that way

the previous line make more sense to me "It's taken me years to settle into this system" , now this is more like it

***carbotaniuman*** I don't think it's half a day to get proficient, it's half a day to hack something half working together so they're unblocked and can do the other stuff they want to do.***throwawaymaths*** And then another half a day to get pissed off because you don't remember what settings are/are not in your venv and trying to exit and/or get back into the venv (assuming no prior experience with venv)

Fundamentally venv breaks your conceptions of what a shell is through cleverness, and that's a problem for people who are new.

***codeflo*** Half a day to get something running that's a "one off" for you is insane. With a compiled language project, I'd download a binary. In Python, I need to reproduce the developer's setup. And I've yet to find two different Python projects where the official build instructions are compatible -- each one recommends a different environment virtualizer, a different runtime, different settings, and different C libraries that aren't part of the virtualized environment.***earthboundkid*** The thing that drives me insane about Python packaging is the degree of Stockholm Syndrome. PYTHON PACKAGING HAS KIDNAPPED YOU. YOU DON’T ACTUALLY LOVE IT.

None of Go, JavaScript, or Rust require a half a day to figure out packaging. Don’t defend Python. Push for it to change or leave it behind.

***mattgreenrocks*** Half a day is an eternity when it is makework.

Trudging through old SO answers, bad docs, GitHub issues, overly enthusiastic blog posts by someone making a todo app, and, the coup de gras: hitting paywalls on the one tutorial that might help you.

Factor in much higher rates for more senior devs and it amounts to a lot of waste.

***Aerbil313*** This whole thread is a discussion of ease vs. simplicity. Easiness is subjective, simplicity is not. We should not compare tools in terms of their ease of use. There was a famous talk on this, maybe someone can link.***kevin\_thibedeau*** I have issues precisely because of the misguided preference for virtualenvs in favor of traditional system package installation. It's obnoxious that pip now admonishes you for installing into site-packages even on a Debian system where that can't cause massive breakage. When you need isolated containers it's great. Everyone doesn't need a webdev focused, reproducible build for everyday shell life.***movpasd*** In my personal experience, it's absolutely necessary. Breaking changes are all over the place. I have non-dev coworkers who have built Python tools without any knowledge of package management, and it's a minefield getting it up and running.

For individualised shell usage, sure. I have global installations of common data science utilities like pandas and jupyter, or requests.

Reproducibility isn't just about deployment, it's also about coordination with colleagues.

***dcow*** You really shouldn’t, though. If you use a dependency manager for some deps you should use it for all deps. Using a global/system cache would be great if dependencies were versioned and each script could specify which version is needed, but they’re not to my knowledge. And it’s all fun and games until some random install script somewhere updates a global dep and your stuff breaks and you don’t know where to even begin looking.***Aerbil313*** I’m still waiting for Nix to mature and for its UX to improve.***pharmakom*** How do you build a bundle for Linux from macOS? In my experience this is only possible with Docker, and is a huge weakness of the Python ecosystem. Would love to hear solutions.***Tainnor*** > I got so annoyed with bundler I wrote a simple shell script to provide the moral equivalent of Python venvs to gem:

This already exists: https://rvm.io/gemsets

Most people don't use it because they haven't felt the need. Bundler already makes sure it loads exactly the right versions of gems for your project, so there's no need to isolate them from each other.

***scruple*** I've used Python intermittently throughout my career but today it is my primary language. I've also never really had issues and have pretty much always stuck with the basics. I'm not on a very large team, though. Only 5 developers will ever touch the Python we're writing. I've only ever worked on Python on very small teams, so I wonder if that factors in?***nologic01*** I think complaining about Python's tooling has become like complaining about the weather in some cold and rainy country. Good for chit-chat and noding in resigned consensus.

In the meantime I cant recall any instance in the last five years (at least) where a project would fail to get going based on a simple venv and pip setup.

An exception is CUDA but I think we can agree this is not what you need to setup every day.

I wonder if that benign view due to Linux and people experiencing something very different in other OS's.

***bradrn*** > I'm always confused about whether I should have ghcup or stack manage my Haskell versions (and what the benefits and drawbacks are)

There was a long and drawn-out transition, but these days it’s quite simple: use GHCup to manage your tooling, and Cabal to manage your packages. At one point Stack was the best option, but no longer: it’s not as well maintained, and is missing a lot of features (see e.g. https://discourse.haskell.org/t/6849/23).

***Tainnor*** You say "don't use stack" and the other person in this comment thread says "use stack". Do you see the issue?***bradrn*** Fair enough, though I’ll note that only I’ve given an up-to-date reference.***Tainnor*** In that same thread that you've linked, other people have later replied arguing for why they prefer Stack so... I don't really think that you've given an argument that is persuasive enough to someone who is new to Haskell.

(And I'm not even that new to Haskell. It's just that I don't use it every day and when I come back to it and have to remember the weird incantations and dances I have to perform to make HLS not crash or want to overwrite my stack-installed Haskell version, I'm usually rather annoyed.)

***pyrale*** > In that same thread that you've linked, other people have later replied arguing for why they prefer Stack so...

I merely stated that I have no trouble running old projects using stack.

> I don't really think that you've given an argument that is persuasive enough to someone who is new to Haskell.

Honestly, cabal improved a lot since the old ages. For a beginner, either stack or cabal should be very fine imo.

> or want to overwrite my stack-installed Haskell version, I'm usually rather annoyed.

Not sure why you would want to do that. Either you use stack and let it handle your ghc install, or you don't, and I really don't see why you would use stack only for compiler install.

***Tainnor*** > I merely stated that I have no trouble running old projects using stack.

I was referring to the linked forum thread.

> Not sure why you would want to do that. Either you use stack and let it handle your ghc install, or you don't, and I really don't see why you would use stack only for compiler install.

What I mean is that when I start VS Code on my Haskell project again after a couple months, it tells me that it needs to donwload a new version of GHC. And I don't know why since I didn't change anything about my project. And when I try to click on update, it fails and the HLS extension crashes. I can't tell you where exactly this goes wrong, but it's obvious that these tools don't all work well together.

edit: Now I see that my wording was confusing. You quoted me as saying that I want HLS to overwrite my GHC version, but what I meant was that I want to prevent HLS (actually, I should have said the VS Code Haskell plugin) from doing that. The subject of "want" in that sentence you quoted is "HLS".

***pyrale*** Ah, didn't run into this issue, as I don't use vscode.

Apparently there is some work being done to improve the stack <> hls experience, but I wouldn't know how it's going and when it's being delivered: https://github.com/commercialhaskell/stack/issues/6154

***nonameiguess*** You posted a link to a forum. The official Haskell language "get started" page says to use both: https://www.haskell.org/get-started/***george\_\_\_\_\_t*** It's listed after Cabal as an "alternative", so a lot of people will presumably ignore it if they don't see any reason why they'd want anything else.

I personally wish that page were even more opinionated, but it's politically tricky.

***Eddygandr*** Stack vs Cabal was a huge argument where I worked with multiple teams using different build tools. Add Nix and nix2whatever, and it was more fun. Spent half my time debugging build instructions***pastage*** It is not that much better in Java land. Getting builds right takes lots of resources, keeping new and old things running is not trivial.***Tainnor*** I haven't had that experience for newer code bases. Yes, there's some ancient "works only in my IDE" stuff but that's really not how modern Java is written.

Yes, there is a disagreement about whether to use maven or gradle, but IMHO they both work reasonably well out of the box.

***twic*** Right now, things are pretty good. But this is surprisingly recent. Gradle and Maven have both been good at managing dependency versions for a long time. Gradle has also been good at managing the Gradle version, via the wrapper, for a long time; Maven has equivalent wrapper now, but it's quite new, and support in the wider ecosystem is patchy (eg a TeamCity Maven build step can't use a wrapper, i don't think). Meanwhile, there is no standard way to manage JDK versions; there are SDKMAN! and asdf, and they work fine, but they aren't de facto standards. Gradle lets you specify the JDK version to build with via toolchains, but this is quite new (6.7 in 2020). I have no idea if Maven has an equivalent.***Tainnor*** While what you say is true, the point about the JDK versions is mitigated a bit by the fact many things are backwards compatible. Although yes, if you jump from 8 to 17, you're going to run into some problems.

I'm not a huge fan of the gradle toolchain tool (I don't think gradle should be managing my JDKs and I would prefer if it could instead just fail the build if you use the wrong JDK version), but I do understand why it exists.

***Mateon1*** In my experience, if you have to build an older maven project, you have to go through a lot of painful hoops, mostly around HTTP vs HTTPS repos and java source/compiler versions. I have encountered many old projects that don't build out of the box, and of those, I've only managed to get about half working.***mrkeen*** Yes. I've been in teams where there has been gradle vs maven arguments.

Right now the Mac (M1 or M2) users in our team can't run our integration tests. It has been suggested that if we migrate our 50-100 services to Java 17 then the tests will work for them again. Trying to do this breaks the Gradle scripts, and the errors are so vague I can't tell why it's complaining.

***Tainnor*** While I understand what you're saying, the "XY doesn't work for mac users on M1/M2" is a really pervasive problem across all technologies.***epolanski*** +1.

I have in most years learned some nicher languages: Elm, Rescript, Racket, Common Lisp.

But nothing is even remotely bad as Haskell is.

Even just setting up an IDE to have basics like syntax highlight or go to definition is a giant chore.

On the official Haskell devcontainer offered by GitHub is nearly impossible to add any external dependency, so having a one-click shareable Haskell environment in the cloud (the very basic to onboard people on the language without wasting them days into having their own inconsistent environment) is a must.

Another thing that I feel sucks is the default documentation for Haskell libraries. Doesn't say anything useful at all, bundles few definitions here and there, 0 examples, and more often than not there's 0 documentation.

***talideon*** I expect that now that things are settling down around pyproject.yaml, Python's tooling will settle down. I've been settling on Hatch for most projects rather than using Poetry these days and Flit for smaller projects. It's managing to replace the mess of makefiles and shell scripts I used to have and supports standardised metadata. These days, I mostly use a combination of hatch, pip-tools (mostly when I'm dealing with lambdas and Django projects), and pipx whenever I'm doing anything with Python. it's not perfect yet, but it's much improved. The new pip dependency resolver still has its issues though if you haven't clamped your dependencies sufficiently and its solver can get caught in loops of excessive backtracking. Hopefully that'll improve too. pyenv is still somewhat hit-and-miss though.

What's really surprised me lately is how much better OCaml's opam tool has gotten! Last time I was using OCaml in anger, the experience was very, very clunky, but these days it's quite smooth.

***pdonis*** > I expect that now that things are settling down around pyproject.yaml, Python's tooling will settle down

First, a nitpick: it's pyproject.toml. YAML was considered for the language of this file and rejected in favor of TOML.

Second, unfortunately, I'm not sure I share your optimism. This is an area where IMO the Python developers have never been able to do things right. In fairness, they are trying to support a lot of very different use cases, but the proper way to do that would have been to let the communities surrounding each of those use cases invent their own tools, while focusing the standard tools in the Python standard library on the vast majority of Python projects that are pure python code--no extension modules, no weird compilation issues, just pure Python modules and packages for which build and install ought to be simple and straightforward. But even that simplest use case was never quite properly and standardly supported.

Even pyproject.toml illustrates this pattern: a new markup language was adopted, one which has no support in Python's standard library, which has no obvious advantages over previous file formats, and which now creates ambiguity between pyproject.toml and the previous supposed "standard" for declarative project metadata, setup.cfg. Which do I use now? I can't just use pyproject.toml because it doesn't include everything that setup.cfg does; and I can't just use setup.cfg if I want my project to be compliant with the latest build tool specs because setup.cfg alone is now considered a "legacy" build format. So now I have to use both, even for pure Python projects where this should have been a solved problem years ago.

At least the actual installation of pure Python projects is now a lot easier; once you have a properly built sdist or wheel, pip install will put it wherever you need. But the "properly built sdist or wheel" part is still IMO a lot harder than it needs to be for most projects.

|||\_\_d||| 100% agree re: TOML. Just ... why? Is there really something so compelling about it that makes it worthwhile being different to everything else?

The fact that the PEP was proposed with this, discussed, accepted, implemented and deployed with this is indicative of the failures of process that lead to many of Python's problems. Python3, async, threads, GUI toolkits, the whole mess of tooling ...

Languages and their ecosystems need someone to say "no". To maintain a pragmatic path forward for users of the language, not developers of the language. Python hasn't got this. With all respect to Guido, and now the core group, they let this slide way too often.

***grumpyprole*** Having developed Haskell professionally for many years, I've spent the last two years programming OCaml professionally. I've been really enjoying the OCaml tooling: dune, Merlin and ocamlformat work extremely well for me. Dune especially is packed full of features, for example a file-watch mode for tests. Merlin doesn't choke on large codebases like some of the Haskell tooling did and ocamlformat works better than any code formatter I have ever used. The tooling is so good, I can forgive the rather minimal standard library. OCaml is also very good regarding backwards compatibility, new releases don't break our builds.***belmont\_sup*** I have to agree so much. Been following for years, and started new experiments this week. It all worked so well.

dune init …

opam install some tools like lsp

restart vscode

dune build --watch and I’m good to go.

Edit: adding some counterpoints and issues to level the field (but are being worked on)

1. dune/opam - wish they were one thing. I can still install packages with dune, and a package lock file exists for deterministic-enough builds.

2. dune files and s-expressions are still weird to write. I wish we can move away, or at best, have a tool that helps you remember and where configs. best I have now is chatgpt instead of having to crawl dune docs

3. eio, lwt, async - 3 async runtimes, hoping that one day it’ll be easy to use an async lib with an lwt-based project via eio. The division will likely never be consolidated

***yawaramin*** 1. I agree. And this is being worked on, in the future dune will take on more package management tasks.

2. Agree, YAML has decisively won the config format wars. But dune it's extremely unlikely that dune will change this, and we can at least tell ourselves that s-exprs are pretty cool after all.

3. Yup, I hope Eio becomes the de facto standard in the future.

***pyrale*** I've used stack+stackage, and I can still run all of my older projects. I agree that as a newcomer the choice is not evident, haskell is a small enough community that finding mentorship is not always evident if you don't know where to look.***mananaysiempre*** > haskell is a small enough community that finding mentorship is not always evident

I don’t know about today, but ten years ago the answer was to hop on IRC and you’d get all of your IRC-sized non-FAQs answered, however difficult they are.

***tome*** It's still the case, as far as I know. Big list of channels: https://www.haskell.org/irc/***bradrn*** I’m not on IRC, but the subreddit, the Discourse forum and the Discord server are all pretty good for this kind of thing too.***pyrale*** There's libera chat now. But not everyone has the idea of looking there.***revskill*** Not quite. I'm not sure about how Ruby with Bundler related to each other.

Recently, i upgraded Ruby, but then what's to do about Bundler then ? I'm confused.

***Tainnor*** Bundler is Ruby's dependency management tool, basically like npm or similar. It's come preinstalled with Ruby for a while now.***regularfry*** Cargo is an evolution of older ideas from Ruby's bundler, so in a sense it does have a longer pedigree than most other tools of that ilk. Bundler mostly doesn't suck. There are design choices I disagree with but there's a happy path to using it that gets a lot right. There are ways to do language tooling that don't suck.

Without much direct experience of cargo I can't say whether that carries across, but I wouldn't be surprised if it was genuinely the best of breed and stays that way.

***saurik*** FWIW, the thing which really sucks about cargo is how it handles cross-compilation. Most "professional" workflows are always cross-compilation: even if you are technically on Linux already you don't want to build for the exact version you have so you create a sysroot and cross compile towards it... and cargo is somehow so bad at this that increasingly large numbers of projects are being forced to turn on a environment variable that has in the name of the variable DO\_NOT\_USE to shift cargo stable into accepting nightly flags so we can activate a patch that they begrudgingly committed to fix the behavior of targets to not get infected by host configuration if they happen to look similar. The goal was that this would eventually become the default but it's been years now and the related bugs are just stacking up. As a package manager I'm sure it's great... as a build system? Not so much.

The situation frankly really just sucks: as far as I am concerned--from the shear number of bugs I have run into trying to do what should be table-stakes, and which even autoconf handles trivially: doing deterministic builds across platforms of a library to be embedded into another language's build system--cargo optimizes for extremely simple use cases (hell: ones so simple they lead to the wrong mental model of compiling in the first place! like, people seem to seriously think you need to compile for old systems ON old systems using old tools as cross-compilation is somehow treated as esoteric) while throwing the people who know what they are doing under the bus, which isn't at all how one should build tooling; instead, the goal should be to make extremely difficult things easier to pull off, even if it means the simple things have to be a bit harder, as your overall workflow is then easier to manage and you can better grow with your tooling rather than having to eventually throw it away entirely.

***kibwen*** You can set a default build target for a Cargo project with two lines of configuration, no nightly features necessary: https://doc.rust-lang.org/cargo/reference/config.html#buildt...

Can you clarify what this is referring to?

> the goal should be to make extremely difficult things easier to pull off, even if it means the simple things have to be a bit harder

There are dozens of existing build systems that have this philosophy, and personally I appreciate the default system being optimized for the average use case. The fact that the happy path is so easy is the reason that people overwhelmingly choose to use Cargo; we can lie to ourselves all we like about the appeal of things like memory safety and type safety, but at the end of the day Cargo is the reason that Rust is popular. (And to be clear, I'm not trying to say Cargo is perfect; I have a slew of my own bugs and feature requests filed against it.)

***saurik*** > ...but at the end of the day Cargo is the reason that Rust is popular.

FWIW, maybe that's true for you, but there are numerous other advantages to the language for which many people choose to use Rust--some even "despite" Cargo: you see Google having had to put in way way WAY too much work to get Bazel working for Rust :/--that it honestly feels a bit like belittling an extremely important language to make this claim so flippantly.

In my case, I often feel bad that I'm not using Rust more often, as I'm one of those developer churning out tons of C/C++ code. I do my best, but... I'm a security engineer! I know what I'm doing is stupid! But like, I'm having a hard enough time just dealing with the Rust library dependencies I've accumulated, so I'm not going to start developing even more Rust :(.

> You can set a default build target for a Cargo project with two lines of configuration, no nightly features necessary...

This doesn't work as, as soon as you start setting target-specific options, it infects the host's options, as they incorrectly modelled the problem as some kind of map from targets to flags. If you don't believe me, on your Linux computer, try cross-compile something complicated that will runs on a "least common denominator" Linux distribution, such as CentOS 7.

> Can you clarify what this is referring to?

Sure. I've Googled rust cargo target host bugs for you (which, FWIW, finds a number of bugs I've filed or have talked about, but it isn't as if I have a list anywhere). Note that one of these bugs is "closed", but I still provide them for context as a patch might have been merged but (as you'll find out if you read through all of these) it isn't stable.

https://github.com/rust-lang/cargo/issues/8147

https://github.com/rust-lang/cargo/issues/3349

https://github.com/rust-lang/cargo/pull/9322

https://github.com/rust-lang/cargo/issues/9453

https://github.com/rust-lang/cargo/pull/9753

The result of this work being left incomplete is that increasingly large numbers of "serious" projects--things I'd expect people in packaging land to have heard of, such as BuildRoot--are being forced to set the ridiculous environment variable \_\_CARGO\_TEST\_CHANNEL\_OVERRIDE\_DO\_NOT\_USE\_THIS="nightly" in order to get access to a flag that makes Cargo sort of work.

I remember a bunch of other ones but I think they are linked from these bugs (I didn't want to dig too long on this); you can also find--either linked from these bugs (as people will tag the bug in their patch) or by searching for that variable (as no one else would use that in a sane project) the projects that are slowly being forced into this crazy workaround.

(What is sad is I often see people surprised at how long it is taking for various of the more important clients to fully get into using Rust, as the safety issues are so severe from continuing to use C/C++; as you made the contention that you believe the reason why people use Rust is Cargo, I will say the opposite: the reason why we don't see more Rust is also Cargo.)

***regularfry*** Yeah, that tracks. That sort of cross-compiling isn't something you'd ever see in the Ruby world, as far as I can recall, so I guess it's a reversion to the mean.***15155*** Does `cross build` not work for your use cases?***twic*** Just do the build in a container modelling the appropriate environment. You don't need cooperation from the build tools (apart from controlling instruction selection), and you also don't need to trust that the build tool has implemented this feature correctly!***saurik*** FWIW, it is easier to just not use Rust as the only serious build system I've had to work with that doesn't support this correctly is Cargo; and like, the mental model you are describing causes all kinds of other problems as it implies that if your build tooling simply can't run on a system old enough that you have to drop support for that system... people who are compiling serious apps (think Chrome or Ubuntu itself) and managing backports aren't sitting around with their CI systems running some old version of CentOS or Debian and praying their build tools continue to run on them: they just cross-compile.

And like, that Cargo doesn't handle this correctly is kind of silly as one of the engineers from BuildRoot seriously broke his back for a year jumping through hoops with them to come up with a compromise and satisfy all of their roadblocks (the Cargo people would rather spend hours talking with people about a patch than spend a few minutes touching it themselves... it was honestly demoralizing, but I've seen the same thing from projects like Flutter, so I think that it is just the reviewer / developer power dynamic that big tech taught a bunch of people) and the code is even in Cargo and yet is rotting in nightly.

(Meanwhile, I often see people bemoan how large projects aren't picking up Rust and using it to slowly cannibalize the C/C++ code in various projects, or are looking at distributions demanding answers for why there has been such struggle getting Rust code distributed, but the answers lie in the build model of the language and in no small part how difficult it is to use Cargo. If you just want to not care about those use cases and want to focus on self-compiled use cases--maybe you are doing web development!--that pick up all of the idiosyncrasies of the build system, then the community should just say that.)

***pyrale*** > so in a sense it does have a longer pedigree than most other tools of that ilk.

That's true of most new languages. It seems like most projects these days can produce a LSP with minimal effort, make the obvious choice of sandboxing every project separately to avoid dependency hells across projects, have at least basic distribution systems even with small communities, etc.

The community learned a lot from the mistakes of yore.

***paulddraper*** All I know about Ruby is that I always seem to download three versions of documentation***nobleach*** Back when I used to use Ruby 10 years ago, I got used to doing the "no docs" args when I'd do a gem install. I always thought it was a cool idea to have local docs but, I never had an occasion to actually USE them. (never was I stuck in a non-internet-connected zone, wishing I had documentation)***regularfry*** Adding `install: --no-document` into ~/.gemrc is something of a reflex for me. Should only be getting `ri` docs these days by default though.***mberning*** I’m a bit incredulous that Java’s tooling is in the “sucks” category. You could say a lot of negative things about Java, but the tools available freely and commercially are in a league of their own.***Tainnor*** I think it's nuanced. IDEs are great, but there's often a lack of good CLI tools and they often show their age. For example, I find checkstyle rather annoying to configure and use.***CraigJPerry*** If you’re looking for alternatives, SonarQube has jumped quite far in the last few releases. I like the coherence of the platform now and i find the default gates aren’t oppressive like they were in days gone by.

I’ve gone all in on them these days, the feedback from sonarlint in the IDE is usually useful.

***Tainnor*** Last I checked, configuring stuff in SonarQube still involves fiddling with the UI instead of config files. And you need to host it somewhere.***pyrale*** Well, it really depends on what you value in your tooling I guess. I tend to appreciate minimalist tooling, and Java definitely doesn't fit that bill.

To be fair, the language needs that tooling to be practical, and the quantity of it isn't helped by its long history so it isn't necessarily the tool makers that are to blame.

***pooya72*** Yes, it's definitely much easier to get up and running with Java than it is with Haskell. This includes everything from setting up the JDK, setting up your IDE, to building and deploying your applications.***dkarl*** What Java build tool would you say does dependency management decently? Most people are still using Maven or Gradle.***Tainnor*** Depends on what you call "decently". Maven and Gradle do the job, but the bigger problem is the ecosystem:

Libraries don't declare version ranges for their transitive dependencies (like they do e.g. in Ruby), instead they just depend on a specific version. Of course, because two different libraries may depend on the same subdependency but in different versions, you will just get one version and you have no way of knowing whether it will work correctly.

That isn't a problem in 90% of cases, but sometimes it is and you'll notice it when you're suddenly getting ClassNotFound exceptions.

There are some solutions to this, e.g. bigger frameworks like Spring publish BOMs, which are just sets of library versions known to work together, but they don't cover everything.

***tadfisher*** You can depend on version ranges with Gradle, and those ranges are transitive if the library is also using Gradle; otherwise, you can override a library's transitive dependencies pretty easily with component metadata rules.

You can do so much with Gradle, but 90% of learning it is figuring out what knobs you need to twist; there's a lot of feature overlap and second-system-effect going on.

***dkarl*** > Libraries don't declare version ranges for their transitive dependencies

That's the behavior that Maven itself recommends, so Maven is at least part of the problem.

Maven could also sanity-check your build for these issues by default, but it doesn't; it leaves that up to plugins, which you have to configure (in XML) in your build file every time you set up a new project.

***pjmlp*** Rightfully so, most of the alternatives are found lacking in feature parity.***paulddraper*** Maven or Gradle***dkarl*** I feel like Maven is doomed by its history. When Maven was created, expressing dependencies with version ranges was encouraged. But there was no lockfile concept, so that did not work well. Instead of adding lockfiles, they decided to leave the tool as it was but encourage people not to use dependency ranges. But dependency ranges are still supported, so you need a plugin to check your POM files and make sure you aren't using them. Dependency resolution is recursive, so indirect dependencies might still use version ranges, so you need a plugin to detect those ambiguities so you can pin those versions yourself in your own POM. If, despite your best efforts, you end up with an indirect dependency being specified with two different versions, Maven doesn't mind that at all (by default, if will package both, and leave it up to chance which one gets loaded at runtime)[0] so you need a plugin to detect that situation as well.

To sum up, we realized the default behavior was wrong over a decade ago, but rather than change the default behavior of the tool, maybe add a new version of POM files where sanity is the default (it's a versioned format! c'mon!) or a new non-XML file format that triggers a new mode of operation, they stuck with the legacy behavior, forcing every new project to include a bunch of XML boilerplate just to get the behavior that they realized should have been the default 15+ years ago.

That was the situation last time I used Maven, anyway. I'd love to hear that Maven has since added a new mode of operation that is sane by default, without needing extra plugins and configuration, and doesn't use XML.

I can't comment on Gradle. I've worked on a couple of Gradle projects, and the build files were a mess, but I don't know if they needed to be a mess or it was just bad luck.

[0] Actually I don't remember if the default is to pick a version at build time or to package both, but I know the latter is possible because I saw it cause many production issues before we configured a plugin to prevent it.

***yw3410*** Also, Mavens metadata model is recursive. You want need a dependency? You're going to have to pull in the parent, which is probably an Uber pom. Take a look at your M2 cache, it's probably got a kubernetes pom even if you've never touched it before because your logging library uses an Uber pom...***TJSomething*** Gradle can be quite good, but it's also really easy to make a completely unmaintainable Gradle file.***twic*** Gradle is expressive and extensible, but unfortunately, that encourages programmers to express themselves and extend it.***paulddraper*** > expressing dependencies with version ranges was encouraged

In years of Java development, I've seen that only once, maybe twice.

And this was 8+ years ago.

***dkarl*** The v1 release of Maven was in 2004. But it's unfortunate that Maven moved away from ranges. The "problem" with ranges was that they created non-reproducible builds, because different dependency resolutions could change the build. Every new release of a third-party dependency had the potential to invalidate a tagged and tested version of your application.

Other build tools in other languages decided to use lockfiles to achieve stable builds with dependency ranges. If your application depends on libraries A and B, and A and B depend on library C, then the build tool can check that the lockfile specifies a version of C that fits the ranges specified by A and B, or, if library C isn't in the lock file, find an appropriate version of library C and add it to the lock file.

But not Maven. Maven's solution is for A and B to declare dependencies on exact versions of library C, and then to pick one or the other depending on many degrees of transitivity separate your project from A or B. Seriously:

"Maven picks the 'nearest definition'. That is, it uses the version of the closest dependency to your project in the tree of dependencies. You can always guarantee a version by declaring it explicitly in your project's POM. Note that if two dependency versions are at the same depth in the dependency tree, the first declaration wins." [0]

So if A depends on C 1.2 and B depends on C 1.5, and A appears before B in your pom file, Maven will bundle 1.2 with your application and not fail the build.

I don't know the history of how anybody ever thought that was a good idea. Anyway, they quickly realized that pinning versions in the build was the right solution after all, but instead of adding a separate lockfile, they decided to make you list all the versions in the project file itself. Which is exactly where you want all your transitive dependencies listed, right in your build file, taking up half a dozen lines each because it's XML, right?

Of course Maven is still happy to fall back its "pick the first nearest" algorithm if you fail to pin one of your transitive dependencies, which means your builds might not be reproducible. My boss (quite sensibly) said our builds had to be reproducible no matter what Maven allowed or encouraged, so I had to write a plugin to check for that.

The real tragedy is that because library publishers no longer use dependency ranges, you get to debug and discover violations of semver yourself. Does library A, which specifies C 1.2, also work with C 1.5? The publishers of library A might know that it doesn't, but they don't publish that fact with their library. Jackson plugins were especially prone to semver-unexpected breakage because Jackson didn't have any stable API for plugins. Jackson plugins typically had to use private implementation details of Jackson to work at all, so they sometimes broke on patch releases of Jackson. Library publishers could have encoded knowledge about this kind of breakage in their dependency declarations, but "best practices" said to specify a single version, so that's what they did.

[0] https://maven.apache.org/guides/introduction/introduction-to...

***jgwil2*** Haskell's tooling has put me off ever really trying to get into it again. However, I've dabbled in Purescript a few times and it always seems to offer a pretty smooth experience, especially if you're familiar with JS tooling.***george\_\_\_\_\_t*** It has been rapidly improving over the last few years, though this can mean it's difficult to keep up.

HLS (the LSP implementation) in particular is a pretty young project, and key compiler improvements which will help it improve are still landing.

***wslh*** I wonder why this discussion about tooling does not include .NET languages or Swift, even if the first class tooling are IDEs such as Visual Studio or Xcode. Most probably because the open source world escapes big tech dependencies?***bluk*** As much as I like Swift, the tooling has been and still is subpar. Swift package management is barely viable.

You have to use Xcode. You can try other IDEs but LSP equivalent features are at the most basic level. Xcode is the kitchen sink of IDEs which has led to many negative opinions as it struggles under its own weight. Every other year there’s new UI for things like debugging which is fine but what would be really nice is if the actual debugging worked. Technically there are reasons why you can’t print a local variable sometimes when stepping through a program, but in practice, I do not care and I want to know what the value is.

If you don’t care about the “optional” tooling like the dependency manager and LSP or a linter (which is closer to ESLint versus Rust Analyzer), the required tooling leaves much to be desired. The compiler sometimes gives up when it takes too much time to process a complex type. It would be understandable in some cases but most people encounter the problem when just writing seemingly simple SwiftUI. Error messages and auto fix-it suggestions are improving but still disappointing. I remember when Apple switched from GCC to LLVM and everyone was praising the error messages as a reason to switch.

Swift is actually ambitious. The generics system is world class. It has to support the legacy of a huge ecosystem on multiple platforms. SwiftUI is one of those bets that you might be surprised that Apple can still make. But whenever I fire up that SwiftUI Preview, I am crossing my fingers that maybe I’ll see something instead of an error. Swift lives up to its name in terms of moving quickly and the language design is probably fine, but outside of that, the tooling is still very immature for a decade old language which tons of resources are invested in.

***agos*** this has been my experience as well. After being one of the five people who liked using Objective-C I was eager to have this new, incredible language to work with, and pushed to adopt it as soon as reasonable on all the codebases I worked with.

That enthusiasm has since waned when stuff happened like compilation times slowing to a crawl because somebody used ternaries extensively. I still think it's a great language, but the tooling really makes it hard to love

***Tainnor*** No idea about .NET, but XCode is a joke of an IDE.***bootsmann*** I appreciate the "run app on any iDevice" with one click that xcode offers, but yeah the rest of the UI is terribly clunky (Try moving the boundaries of frames).***Tainnor*** Not just the UI. The IDE barely has any functionality. Automated refactorings? Nope.***pyrale*** That's simply because I talk about my personal experience, and there's only so many languages you can experience professionally over a few years.***paulddraper*** Linux***wslh*** Yes, but you can run both in Linux natively.***lovasoa*** > I simply wanted to filter job offerings based on social utility rather than language stacks

I wish more people did that.

***ParetoOptimal*** In my experience the organizations that claim to have social utility fail at it and don't pay well.

Language stacks are much easier to fact check.

***pyrale*** You don't have to listen to orgs to make your opinion.

There are plenty of orgs that obviously produce a vital service for society, and they are usually starved of good IT people: health care, social services, logistics, electricity production & distribution, industry, agriculture, emergency responders, etc.

> and don't pay well.

:( Well, the comp is still high enough to live a comfortable life, I just won't be able to bankroll a political party in the near future.

***Symmetry*** I good place to start looking:

https://jobs.80000hours.org/?int\_campaign=job-board

***Eddygandr*** I think this happens with experience - I’ve recently done the same after spending years chasing languages to work in***a\_random\_canuck*** I too have used several languages and for me Haskell’s is the worst I’ve had the displeasure of using. It could be forgivable if the language itself wasn’t so unfathomable to newcomers.

JavaScript’s situation isn’t great, but it’s still easier to work with than Haskell, in terms of hours of frustration.

For me it’s Rust that has really raised the bar for what quality tooling looks like.

***swader999*** I haven't seen better tooling than what's available for C#. Curious to hear from others who strongly disagree.***all2*** C# tooling is dense, but the build tools generally "just work" TM. Occasionally I have to purge my build artifacts because the build tool doesn't clean things up completely, but that is a small pain compared to working with python. Another commenter noted that Poetry is quite good and it is (I've noticed they added venv support directly into the tool - you still need pyenv to manage multiple venvs, though).***alex\_smart*** What is wrong with Java's tooling?***pyrale*** If you ask for my personal experience:

- The community leans a lot on configuration over code, and that's annoying. Sometimes, a hardcoded string in your conf could have been a hardcoded string directly in the code.

- Sometimes, dependency injection systems are so abstract that knowing which class is depended on in a specific runtime instance becomes a pain in the ass.

- Your idea just won't load class x, the obvious "clean cache" doesn't work as advertised, and you're back to stack overflow to know which couple of file you need to get rid of. That stuff happens just often enough for you to have a lingering feeling of annoyance, but not often enough for you to remember the exact files.

- Sure, idea is great. But in <other language> I was fine with just vim. The LSP was a great help, not a lifesaver.

- Unit testing is great. Annotations are great. Don't you enjoy that unit test class with a dozen annotations spanning 20 lines just above?

- Hibernate. Spring too, while we're at it. Not sure whether you consider this one squarely fits the tooling box.

***systems\_glitch*** These are job security features :P

Seriously though, the hours spent fighting dependency injection and Hibernate issues alone, when working on a really big Java project, could've been a full-time job.

For maximum fun, I once worked on a large ERP system that started out as a Struts 2/Hibernate 3/Jetty project and had an entire Rails app bolted on using JRuby. Some of the stuff in the JRuby side was injected thru Spring. ActiveRecord had to talk through Hibernate.

***patmorgan23*** Gross***systems\_glitch*** I know, right? Hundreds of millions of dollars flowed through that system every year. I got the ball rolling on killing that design with fire after we hit one of the pathological limits in IIRC Hibernate 3 and thread safety.***Tainnor*** I agree with some of that criticism (although I haven't had issues with having to manually delete files - if you do, I suspect it's a project that hasn't been set up with a proper maven/gradle setup but where all the build info is in some IDE config. that's an antipattern at this point but used to be very common).

But I don't think it's necessarily about the tooling.

And yes, Hibernate is just horrible.

***pyrale*** Everything is in Maven and works fine in CI, which definitely does not use idea. But in some situations Idea believes the pom isn't as fresh as its rendition, unfortunately. I can live with some caching, but everytime that happens and the "clean cache" option doesn't work, I weep.***MBCook*** I don’t tend to run into that. But you should be able to fix it right clicking the pom and choosing Maven -> Reimport.

The number of times I actually needed to clean my cache I could probably count on one hand over the last 10 years.

***gmfawcett*** Really, only the LSP comment relates to Java tooling. The other comments are about language, framework, and cultural issues. Not disagreeing with your observations, though.***xmcqdpt2*** Too much XML?

Seriously though what's a decent Java build tool? Hacking on Gradle means having to learn another PL entirely. Maven?

Then say I want to publish a library for others to use from their own java project, how do I do that? I've never actually done it, but that page

https://maven.apache.org/repository/guide-central-repository...

seems awfully complicated compared to say

https://doc.rust-lang.org/cargo/reference/publishing.html

or even the pip equivalent.

Granted, working on a well set up java project is nice, but the setup process is not simple.

***chriswarbo*** I actually prefer using Maven and its giant XML files: at least they're declarative, and are easily parsed, transformed, generated, etc. by scripts. Most attempted replacements (Gradle, SBT, etc.) stick with largely the same model (i.e. no extra functionality) but use a full programming language for their "config" (Groovy, Scala, etc.).

The latter gives us a "config" that's subject to Rice's theorem: it's essentially opaque, with no way of knowing what it will do other than executing it. An example I ran into at work: there's no way to list the dependencies of an SBT project (in order to set up an offline sandbox, in our case for reproducible building with Nix). SBT provides commands which claim to do that, but config files often append dependencies based on arbitrary logic; e.g. we had some like "when running unit tests, add this mocking plugin"; since "list dependencies" doesn't run the unit tests, that plugin dependency was missing from the sandbox.

I can't speak to the "publishing" situation for Java, I don't have any experience with it. All of our projects transparently pushed/pulled via a Nix cache, whether we used Java, Scala, Python, NodeJS, etc.

***Tainnor*** > there's no way to list the dependencies of an SBT project (in order to set up an offline sandbox, in our case for reproducible building with Nix)

Well, the same is true for Maven. I know because I've tried. Plugins can download arbitrary dependencies at execution time.

That's where the point about Rice's theorem falls apart: it applies as much to maven as to Gradle because maven plugins can do whatever they want.

***chopin*** But that can't be runtime dependencies to your code?

I've written a couple of Maven plugins and you must declare your dependencies explicitely even for those. Pulling in stuff dynamically would be possible but not very clever.

***Tainnor*** The maven surefire plugin downloads dependencies dynamically depending on the kind of tests it finds. Is it not very clever? I agree. But it's one of the most popular maven plugins.***chriswarbo*** That sort of thing is unfortunate, but not too bad since we can add those packages to our own dependency list (e.g. in the pluginManagement section). It's a bit redundant, but as a bonus it ensures that such "dynamic" behaviour is acting as expected (since the build will fail if it picks different packages to the ones we wrote down!)

The same approach can be used when depending on third-party jars/plugins which don't fully specify their dependencies: just add the missing ones as extra dependencies of our project. (This happens a lot, where projects have undeclared dependencies on some commonly-used library, and don't notice since it's usually available in a well-stocked ~/.maven2 cache)

***chopin*** During Maven build, sure. But it does not add unknown dependencies to your code at runtime.

I am used to classpath problems with surefire but that it downloads undeclared stuff is new to me. Afair it pulls only in transitive dependencies. Could you give an example?

I am mainly building war files and it turned out to be good practice to explicitely declare any ambigous transitive dependencies. There's even a Maven plugin which shows up classes with different hash during build, https://basepom.github.io/dependency-versions-check-maven-pl...

***Tainnor*** "Runtime" was imprecise, I meant at build execution time. The point being that you can't pre-download the dependencies without executing the build.***grumpyprole*** Yep, SBT and Gradle make Maven look good.***twic*** Gradle is declarative. The 'code' you write does not conduct the build; it assembles an object graph which describes the build, much like the one you get from parsing the Maven XML. The advantage of using code for this is that you the description can be more concise, expressive, etc. But once it's run, the graph is built, and you can explore that, list dependencies, etc, quite safely.***grumpyprole*** > Gradle is declarative

Gradles own docs say "Well-designed build scripts consist mostly of declarative configuration rather than imperative logic".

When it's up to the programmer to make a script declarative, it's not a declarative language.

***alex\_smart*** That maven link is perhaps not the most useful, but it is not hard to find better-written articles. For example-

https://docs.github.com/en/actions/publishing-packages/publi...

The steps are roughly: 1. Configure the repository you want to push to. 2. Set up your account with the repository. 3. Configure your credentials. 4. Deploy.

***pjmlp*** Not only do Maven and Gradle support binary libraries (including native code) instead of waiting to build the whole universe on each checkout, their plugins for mixed language development are way better than writing build.rs scripts.***gmfawcett*** Yes, Maven. Just embrace it, and move on: the decision will pay dividends.***paulddraper*** So you don't like a config language like XML because \_\_\_\_\_ and you don't like a DSL like Groovy because \_\_\_\_\_.

What is acceptable?

P.S. For easier publishing, use a different repo e.g. Artifactory instead of Maven Central.

***layer8*** Maven is awfully slow and awfully stateful (plugins or multiple executions interfering with each other), has too many quirks, and documentation is lacking. Gradle is imperative and also stateful.

Dependency injection and annotation-driven development is “magic happens here” that is hard to analyze when something doesn’t work, and hard to reason about in the sense of building a proof that it will always behave in the correct and intended way.

***alex\_smart*** > Dependency injection and annotation-driven development is “magic happens here” that is hard to analyze when something doesn’t work

I don't quite get the criticism against annotations. Metaprogramming and code as data seem to get a lot of love in the context of lisp, python or ruby, but a lot of hate when it happens to be done in Java. I absolutely love the powers that reflection, annotations etc give me.

The difficulty of analyzing dependency injection and "magic happens here" is overstated. Just run the code in the debugger. And if you are having problems with the auto-configuration stuff, you can always selectively disable it and explicitly define the beans you need.

***layer8*** > Just run the code in the debugger.

How do you set a breakpoint on an annotation? You can’t, and that’s the issue. You can’t reason about annotations in the precise way you can reason about library calls.

***alex\_smart*** > How do you set a breakpoint on an annotation? You can’t, and that’s the issue.

You can set a breakpoint on your normal method and you can see if any proxy object was introduced by the annotation in the call stack and set a breakpoint there. You can also just see the usages of the annotation to see where and how it is being used and set a breakpoint there.

Anyway, going back to my original question, similar criticism is equally valid when you do metaprogramming in lisp/python/ruby. Why are these concerns only raised in the context of Java?

***grumpyprole*** Don't forget Scala's SBT, an awful design with lots of additional bad taste added on top.***friend\_and\_foe*** Yeah most tooling sucks. I think tooling in general was a bad idea. A compiler and an interactive shell/interpreter are all you need, you look at NPM and the current state of the python package management system, a total disaster. I personally would rather use a language where I had to include all dependencies within my program, rather than automatically fetching them from a remote server, deal with versions and version locks, all that stuff was a bad idea.***pshc*** Python and JavaScript both had their various fragmented package management systems added well after the language was invented and suffer for it. JavaScript is especially hobbled by the need for an intermediary bundling step.

More modern languages like Rust, where tooling was carefully designed in tandem with the initial language release, fare much better in this regard.

***throwawaymaths*** > A compiler and an interactive shell/interpreter are all you need

You might also want a good testing in the lang or stdlib, and a documentation system.

***friend\_and\_foe*** Well the thing about testing with Haskell, the type system ensures that you're not going to have the most common bugs. If there's an error, it's probably an error in your logic or you did some math wrong or something like that. So testing, you'd basically write tests for your mental model of how the program should operate. And because of the type system you're probably going to use unsafe for a lot of test code.***throwawaymaths*** Clearly the type system is comprehensively self documenting too.***mymac*** If only Python would be able to really solve their dependency and backwards compatibility issues, those are really holding the adoption back. Though there is a good chance that even if they fixed those that people burned in the past will never go back into it.***boxed*** > [python] backwards compatibility issues

What issues? A lot of problems with Python is due to keeping compatibility with Python 2.0. Implicit string concat bites me fairly often for example, and it has never been useful.

***mymac*** 2->3 has been a complete disaster, anything older than a few weeks tends to randomly break with some kind of dependency issue, sometimes requiring multiple installations of python on the same machine which will bite each other in hard to predict ways. Python is a wonderful idea but I've yet to be able to write something in python and call it 'finished' because it never ever continues to work in the longer term. Highly frustrating and in my opinion unnecessary.***boxed*** > 2->3 has been a complete disaster

It WAS a long slog yes. But now it's pretty much done. And there really was no way to fix the unicode issue without a big painful transition.

> anything older than a few weeks tends to randomly break with some kind of dependency issue

I absolutely do not have this issue. Maybe you're using libraries very different from what I do? But I do think I have pretty wide interests/projects...

> sometimes requiring multiple installations of python on the same machine which will bite each other in hard to predict ways

I don't know what you're talking about here. Do you have an example?

> I've yet to be able to write something in python and call it 'finished' because it never ever continues to work in the longer term.

I don't have this experience.

***paulddraper*** What dependency and backwards compatibility issues does Python have?

That other languages don't?

***mymac*** Python software simply rots while you're not watching it. Either you make it a full time occupation, every time some library gets an 'upgrade' (with a ton of breaking changes) you get to rewrite your code, sometimes in non-obvious and intrusive ways. And every time the language changes in some breaking way you get to spend (lots of) time on debugging hard to track down problems because the codebases they occur in are large enough to mask the problems that would have come out if the same situation had occurred during development.

And that's before we get into the various ways in which python versions and library versions can interfere with each other. You couldn't have made a much bigger mess if you tried. And I actually like the core language. But so many projects I wrote in python just stopped working. I remember having a pretty critical chunk of pygame code written for a CAD system that just stopped working after an upgrade and there was no way it was ever going to run again without rewriting it. That's the sort of thing that really puts me of an I remember it long after. Machine learning code is still so much in flux that it doesn't matter. But hardware abstraction layers such as pygame should be long lived and stable between versions. And that really is just one example.

Anyway, I think asking 'That other languages don't' doesn't really matter. But Haskell (see TFA) is one language that always tried hard not to be successful so breaking changes would be permitted (which is fair). Python tries both to be popular and to allow for major stuff to be broken every now and then and that is very rough on those that have large codebases in production.

By contrast, COBOL, FORTRAN, LISP, ERLANG, C and C++ code from ages ago still compiles, maybe you'll have to set a flag or two but they're pretty careful about stuff like that.

***aidos*** Are you pinning your dependency versions? If so, things should all still work later.

If you upgrade libs then sometimes you need to do some work. I’ve found python libs to be pretty stable though so it’s never too bad.

***dboreham*** > Either you make it a full time occupation

LOL Javascript enters the room.

***otikik*** > filter job offerings based on social utility rather than language stacks

Any pointers on how to do this? A person that close to me is thinking about entering the High Frequency Trading world, and I would like to give them some alternatives.

***RhysU*** Pushing advertisements to people who don't want to see them? /s

Every trade matches two parties who came to the market to trade.

***otikik*** > Pushing advertisements to people who don't want to see them? /s

That’s… where they work at now

***pyrale*** Well, for starters you have to fight the urge to get as much compensation as you can. Once you realize that 90% of SWE job offerings will land you in the top 10-20% of earners, and provided you're fine with the lifestyle that allows, most companies need tech.

The two criteria I'm interested in as an IC is how useful the job is in my eyes (this you can know before applying), and whether the company gives me enough independence to be able to make things right. So my interviews focus on a single factor: whether the company culture and organization is conductive to good, productive work I will be proud of down the line.

***ryukoposting*** In terms of tooling, Haskell has one thing that AFAIK no other language can compete with: Hoogle. Hoogle is amazing. You tell it, in Haskell, what you want, and it tells you, in Haskell, what you can do. It's extraordinary. Someone attempted something similar with Rust, and I even tried to make a Noogle (Nim), but it just doesn't work the same in languages where there's a clear divide between "passing arguments to a function" and "calling a function." I find myself looking at tangles of Rust code with all its Result<Option<Box<SomeEnum<Nonsense>>>, Box<dyn Error>> and I yearn for a Roogle that provides the same level of utility as Hoogle.

Other than that, Haskell's tooling has no redeeming qualities. Nothing (pun intended). A lot of that can be blamed on the community's instinct to make something "innovative" instead of improving what already exists. I feel the author's pain.

***runeks*** Hard agree on Hoogle — it's amazingly useful.

But wrt. other tooling, I use haskell-language-server every day and it makes me so much more productive. Sure, it isn't perfect, but so much better than what we had just five years ago.

***Dobiasd*** Hoogle is really amazing!

Inspired by it, I implemented something similar for FunctionalPlus (a functional-programming library for C++): https://www.editgym.com/fplus-api-search/

I'd love to see more projects taking this path too. :)

***runeks*** > The way that Haskell-the-language evolves — well, the way that GHC evolves, which is de facto Haskell since it's the only reasonable public implementation — is that it gradually moves to correct its past missteps and inconsistencies even in pretty fundamental parts of the language or standard libraries.

I would say that the biggest problem is that GHC is tied to a particular version of base (the standard library). So when changes are made to base, and a new version of GHC comes out that supports only this and not earlier versions, you're forced to change basically all of your dependencies, if you want to use this newer GHC version, as they all depend on base.

I still don't understand why this is necessary. Why must code compiled with GHC 9.6 use base version 4.18.0.0? Why should the binary that is GHC care about which version of the Data.List module the code that it compiles uses? I understand that all the GHC-specific stuff exposed by base is tied to a particular GHC version, but why all the rest?

There is, however, work in progress to split base into multiple packages to fix this (as I understand it): https://gitlab.haskell.org/ghc/ghc-wiki-mirror/-/blob/master...

***mrkeen*** > I still don't understand why this is necessary. Why must code compiled with GHC 9.6 use base version 4.18.0.0?

It's hinted at in the section you quoted. A newer ghc might reject older base code as invalid.

***runeks*** That GHC might reject an older version of base isn't a reason to switch for every GHC release, is it?

I mean, if it breaks then sure, require a newer base. But in my experience GHC (thankfully!) doesn't change the semantics of Haskell often enough to warrant a new version of base for every new GHC version.

***tinco*** > I still don't understand why this is necessary. Why must code compiled with GHC 9.6 use base version 4.18.0.0? Why should the binary that is GHC care about which version of the Data.List module the code that it compiles uses?

Because the underlying data types might be different, so if different libraries linking to different `base` implementations pass each other instances of `Data.List`. Imagine for example a Data.List Data.List, could you append the results of functions out of two different libraries to that list?

***runeks*** I'm not suggesting that my library should be able to transitively depend on multiple versions of base.

I'm suggesting that which version of base my library depends on should not be tied to what the GHC version (used to build my library) depends on — unless my library is using the GHC-specific stuff in base.

***tinco*** Oh yeah, I think that's just a convenience thing. New versions of base probably use new features of GHC, so you'd end up with a backwards compatibility matrix. People would get angry either way, and I guess it's more convenient to not put in the work and just have people angry all the time so no one can say they've been misled about GHC's stability.***sclv*** Indeed, as you note, a reinstallable base is a goal everyone wants. Its basically historical reasons and coupling of primitives (tied to the compiler innards) to nonprimitives which has caused this situation, but sufficient elbow grease should improve things.***pooya72*** If I had to choose the three big factors that contributed to my gradual loss of interest in Haskell, they were these:

\* the stylistic neophilia that celebrates esoteric code but makes maintenance a chore

\* the awkward tooling that makes working with Haskell in a day-to-day sense clunkier

\* the constant changes that require sporadic but persistent attention and cause regular breakages

Valid points. Back in 2010-2012, I spent a lot of time learning Haskell. The language itself is great, but the documentation and tooling was challenging to work with. The community went from Cabal (and the infamous Cabal hell) to Stack, and back to Cabal. Overall, the situation has improved.

On the other hand, other programming languages have incorporated elements of functional programming. Take Java, for instance. It has added features like Streams, functions, lambdas, algebraic data types, records, and pattern matching. While Java's syntax isn't as elegant as Haskell's, it does include the fundamental concepts of functional programming.

***pyrale*** > Take Java, for instance. It has added features like Streams, functions, lambdas, algebraic data types, records, and pattern matching.

In a doomed attempt to escape the prison in which they were locked, the inmates defiled their language and adopted grotesque rituals inspired by the light they saw through the bars of narrow windows. They created an endless pit of suffering of their own, which is made tolerable only because the light shafts are too high for them to see the colors created by light on trees outside.

Those that came from outside quickly lose sanity, constantly being by torn between a dialect adapted to self-imposed darkness, and a dialect that could thrive in the light but is stiffled there.

***nobleach*** Perhaps, but what you may not understand is that not ALL developers \_want\_ a purely functional language. For some, things like Kotlin hit a sweet-spot. One can lean a bit more into a functional style, or they can lean more into an OO style and it's acceptable. Some are very interested in thinking in terms of Functors, Applicatives, Readers, etc... some just want map/filter/reduce. That's what the Streams API did for Java devs. For me, Haskell isn't "the goal". It's simply one way of solving problem-sets.***pyrale*** The meaning of my message wasn't that everyone should move to a functional language, but that Java devs shouldn't. The language has some decent features. Under all the OO patterns and abuses of framework, there was a decent core to save. My issue is that the community has declined to write their own "Java - the good parts", and tried to bolt half a dozen pair of wings on their supertanker because planes are faster than boats.

As a java-turned-haskell-turned-java dev, I can enjoy some OO programing, even though I prefer FP, but I definitely don't enjoy unprincipled FP riddled with side-effects, null pointers and built upon the quicksands of frameworks that are thoroughly unfit for that purpose.

***Tainnor*** Kotlin doesn't get enough love. It gets derided by some Java developers for being too cutesy and sugary and it's not talked much about by the kinds of people who love to talk about Haskell, Lisp or Rust (no shade to these languages), but to me it's the most pragmatic language I've used so far.***dkarl*** Speaking as a lover of Scala who has seen Scala codebases go wrong in the cliché ways, Kotlin would be the first language I would consider if I were starting a new commercial software codebase today. Kotlin seems to have enough of the pragmatic elegance of Scala to get the job done with clarity and accuracy, without the curse of attracting the compulsive intellectual thrill-seekers that will ruin your Scala team if you accidentally hire one.***nobleach*** Compulsive intellectual thrill-seekers... while I never want to call names, this does seem to explain the phenomena quite well. There are just some people out there that need the world to know how smart they are.

Anecdotal: I do know one guy however who's just so ecstatic every time he figures new things out. To his way of thinking, everything that comes with "functional" truly is simpler. He built a dependency injector based on a Reader. He built a cool Result library where he managed to get at the internals of the JVM. Never once did he come off as holier than thou. I can appreciate and respect that.

***dkarl*** IMO "compulsive" is the damning word there. People who don't enjoy learning typically don't make good programmers, but you have to show respect for your coworkers when you choose your learning opportunities.***xwowsersx*** This is exactly how I feel, having written tons of Scala, some Haskell and a multitude of other languages. It's why I recently decided to build most new, large projects with Kotlin at $job***LinXitoW*** I really liked Kotlin in the beginning, but it's error handling story is non-existent. They removed the ONLY decent (not great, very flawed) complier validated method for error handling, checked exceptions, and gave us nothing in return.

It'd be so easy to simply do exactly what Rust does. Kotlin already has the ADT stuff necessary, all we need is some kind of "bubbling" operator, like the ? in Rust. Because right now, it's all just super horrible manual checks everywhere.

***Tainnor*** Different strokes... I think Java checked exceptions are a design flaw and the fact that no other language that I'm aware of has them is telling. Maybe it even was a good idea in the beginning, but it was just horribly abused by basically every library under the sun - no, I don't want to catch some low-level exception when I'm calling your library, thank you very much.

I don't mind making error handling part of compile-time checking (as long as you can opt out by something like unchecked exceptions), but that should be done in a way that is compatible with the regular type system. That way you can abstract over error handling in ways that using checked exceptions disallows.

IIRC, the stance of the Kotlin team is that if you want compile-time checked error handling, you should use Result types. Maybe the ergonomics aren't as good as in Rust, but if you use something like arrow, they're still decent.

***paulddraper*** > Kotlin doesn't get enough love.

Really???

It's probably the #1 language for new Android projects.

***xwowsersx*** I presume they meant outside of Android development.***Tainnor*** Yes. Many people say that Kotlin is only for Android but that's totally untrue.***AtlasBarfed*** Have you ever used groovy re: pragmatic language?

I've tried a bit of Kotlin, but still prefer Groovy by a lot.

***LinXitoW*** As someone that has done some Haskell, and does Java for a living, I think their metaphor is extremely accurate. It's not about purity, it's about how the functional parts were grafted onto the language, and therefore don"t "interop" with the classical ways very well.

In a good mixed paradim language like Rust, you can freely choose the correct paradigm for the problem, and can easily mix them. In Java, they are often at odds with each other. Best example is exceptions (esp. checked exceptions) and all Stream operations. They take a lambda which does NOT throw any exceptions. So, you need to either never use checked exceptions (which is impossible, because most libraries still do) or not use streaming, or create a horrendous hybrid.

***nobleach*** This is a very solid counterpoint to my argument. The no throw in a lambda is a fairly nasty wart and should have been handled in a more conformant way.***yw3410*** To be fair, mixing higher order functions with effects (exceptions) is really hard without changing the type system completely. At that point you're moving into novel encoding territory.***mrkeen*** > Perhaps, but what you may not understand is that not ALL developers \_want\_ a purely functional language.

The ability to use pure functions is one of those hills I'll die on. Hearing that not all developers want a purely functional language is like hearing not all surgeons want to wash their hands, or not all accountants want to use ledgers.

***harry8*** yeah sure, maybe, but when I look at software i can install which has a purpose other than programming a computer there's a metric f.ton() written in Java (a language I don't like much) and we can count the items on our fingers written in haskell (a language I greatly prefer for aesthetic reasons).

Xmonad, pandoc and there are more. Let's list everything we can.

We can scream at anyone who points this out or face up to it and work out /why/ and how to actually /fix/ that issue.

When i mention it here it's about 50/50 which way it goes.

***goto11*** Weird then, that amazing software like Minecraft is written in Java, while Haskell seem to be mostly used for writing monad tutorials.***cryptonector*** Funny to call voluntary insane asylum residents "inmates".***freilanzer*** Poetic truth.***dgb23*** Efficient immutable data structures are more important for FP than most of the features you listed. What is Java providing there?***xwowsersx*** > The community went from Cabal (and the infamous Cabal hell) to Stack, and back to Cabal.

I didn't know this. I've been away from Haskell for a couple of years. When I last used the language, Stack seemed like smoothest experience and solved many of the pain points with Cabal. The community went back to Cabal? What did I miss? :)

***consilient*** > The community went back to Cabal? What did I miss? :)

Cabal got better, stack stayed the same. It's more a cultural divide than anything at this point.

***aetherspawn*** I currently don’t use Haskell but maintain an old Haskell package from a failed startup of mine that has some commercial usage by other orgs. I have been assuming everyone is using stack? I’m also interested to hear what is going on here***aetherspawn*** Just came here to say that I re-installed Haskell on a new ARM MacBook and the tooling is fine ... everything setup in about 10 mins and everything worked just fine.***AnimalMuppet*** I mis-read that as "stylistic necrophilia", which... rather changed the meaning.***JackMorgan*** Many of these reasons are why I moved to F# and haven't looked back (much). I sometimes miss Higher Kinded Types, but F# still has generics and if I'm being honest it forces me to write even simpler code then I would have in Haskell. I generally prefer this outcome.

However F# never feels "leet" like Haskell does. It's like Haskell, but all business. I get a lot done in F# and really enjoy it, but I'll catch myself looking back wistfully, like Haskell why couldn't we make it work. Sigh, the one that got away I guess.

***xupybd*** I'm glad to read this. I'm new to FP and enjoying F#. Because it's terse and down to business. I've often wondered if I'm missing out by not using Haskell. It seems, for my purposes, probably not.***Shumer666*** First off, learning Haskell's like trying to decipher an alien language. If you're used to plain ol' if-else loops and straightforward variable assignments, prepare to have your brain twisted into knots. Haskell's got monads, and no, they're not some new type of space monster – they're these weird abstract things that'll leave you scratching your head and questioning your life choices.

Now, I know we all love libraries that make our lives easier. But with Haskell, you might find yourself on a treasure hunt for a library that actually does what you need. The Haskell library scene's like a half-empty thrift store – you gotta sift through a bunch of outdated, half-baked options before you stumble on something that kinda works. And don't get me started on documentation – it's like reading hieroglyphics half the time.

Oh, and performance? Sure, Haskell's got that reputation for being all slick and optimized. But in the real world, you might end up scratching your noggin over why your code's chugging along slower than a snail on a summer day. Lazy evaluation sounds all fine and dandy until your app's gobbling up more memory than it should and moving slower than molasses in January.

Let's talk job prospects, shall we? Unless you're hoping to work on some super niche project for a company that's all in on Haskell, you're gonna have a tougher time finding a gig than a polar bear in the Sahara. It's like showing up to a party where everyone's talking about the latest celebrity gossip, and you're there with your collection of 19th-century poetry – cool, but outta touch.

And let's not forget debugging. Imagine trying to find a needle in a haystack, except the needle's your bug and the haystack is a jumbled mess of functional hieroglyphs. Good luck trying ^^

***rendall*** Definitely worth repeating! https://news.ycombinator.com/item?id=37251323***kajumix*** I have been using Scala. I have found that it can give you the best of both worlds. You can reason algebraically, and often after a refactor, if it compiles, it works. Type inference and monads works great too. You also get to benefit from being in the java ecosystem. In instances where it gets too esoteric, I can break rules and code more like java. I am curious what others think.***inickt*** I have really enjoyed Scala for the same reasons. Has some escape hatches if needed, and a large ecosystem.

sbt drives me insane though, and is probably my least favorite part of the development experience.

***jmaker*** Opposite experience with sbt here. It’s been much more useful than Maven or Gradle. Only fewer plug-ins than Gradle.***kajumix*** I share the hateful sentiment. Too much magic and mystery in sbt for me***valenterry*** Agreed, I feel the same. It also allows for a smooth gradual transition from imperative/oop code to a pfp style.***xwowsersx*** It seems to me that the stewards and maintainers of the language actually \_intend\_ for Haskell to be friendly to research, experimentation and academic pursuit. That is fine, as far as it goes, but obviously this will, at some point, be at odds with the interests of programmers looking to use Haskell as a practical, stable tool. It sounds to me like what is needed is the ability to mark all the experimental, envelope-pushing bleeding edge stuff to a different track (whether through pragmas or even a package level declaration) so that programmers know just by looking at the package whether it's something they want to pull in. This would allow practically-minded developers to adopt a policy along the lines of "we only use Haskell stable" or whatever it'd be called. The dynamic I'm describing is the "Avoid success at all costs" phrase, right? The idea being that if Haskell adoption gets "too high" then the language will become inertial and be unable to continue pushing bleeding edge concepts. What I'm proposing is a way to allow that to happen but also maintain a separate track at the level of the language stewardship that formally acknowledges and recognizes that day-to-day programmers need some way to opt out of some of the edgier stuff and to stay within a more limited subset of stable Haskell.***DonaldPShimoda*** > It seems to me that the stewards and maintainers of the language actually \_intend\_ for Haskell to be friendly to research, experimentation and academic pursuit.

Yes, this is explicitly stated as a core principle in "A History of Haskell: Being Lazy with Class" (2007) (direct PDF link: https://www.microsoft.com/en-us/research/wp-content/uploads/...).

***mrkeen*** > That is fine, as far as it goes, but obviously this will, at some point, be at odds with the interests of programmers looking to use Haskell as a practical, stable tool.

That's what Stackage is.

Stackage provides consistent sets of Haskell packages, known to build together and pass their tests before becoming Stackage Nightly snapshots and LTS (Long Term Support) releases. [1]

Java will never get this.

[1] https://www.stackage.org/

***BellsOnSunday*** There is the Simple Haskell initiative, which encourages what you're talking about, but no flag or pragma that says "this project uses simple Haskell". Obviously, simplicity is in the eye of the beholder. Fancy type features do have their use cases where they make types more expressive, the code safer and even simpler, so long as you've internalised how they work.***joeyh*** This is a pretty good post. The weak part of it to me is that I have never felt pushed to use any particular new fancy type stuff if I don't want to. Don't want servant's type-level http apis? Drop down a level and use warp. Libraries are often layered this way because it's understood that excessively complicated types can be a trap. One does have to develop an intuition for how far to go, which will involve making mistakes.***cflewis*** Do you know of a good "I don't know Haskell well but this Servant idea sounds incredible, please educate me?" article? I read the Servant docs and they (rightfully) assume I have more Haskell understanding than I do :(***mrkeen*** https://www.well-typed.com/blog/2015/11/implementing-a-minim...***mattgreenrocks*** This really resonates with me.

I’ve been using it in a decidedly industrial application for about 1.5 years now. I had some fairly significant experience with it prior (https://github.com/mattgreen/hython).

For the first time in a long time (20 years experience) I’ve needed to learn a significant amount of things. It’s a combo of the domain and the language. It’s rather exhilarating, and also exhausting. Could also be a lot to bite off on with a busy home life too.

Regardless, the language is brilliant. My manager exhorts me to generally write in a top-down manner a lot because Haskell’s flexibility really conveys dev intent well, so think hard about how it should read, and start from there. This is a huge mindset shift from most langs, where you can feel your brain shut off to save cycles as you type “function” over and over. It really feels like it is meant to be write-friendly. Point-free functions are wonderfully terse to write. I joke that TH is my favorite language: a type-checked macro language that lets me write almost anything I want.

And there’s the rub: even with controlled effects via monads, the syntax is still hard for me to scan and read. I don’t know if this comes eventually or what, but this feels like a function of how dense a line could be. I miss early return dearly, and understand why it isn’t a thing (except if you have a MonadZero at hand) but I know it’s a syntactic transformation that won’t make it in. I really miss the amazing Rust LSP. Haskell’s recently lost the ability to flesh out pattern matches due to Haskell internals shifting with 9.x. I still hate and screw up stacking monads. Compile times can be brutal, esp if you hit the lens library. Finally, I’m not a big fan of pervasive laziness: the community has sort of admitted that Haskell programs are far more prone to space leaks developing from this default to the point that many programs may have them go undetected for quite awhile. The systems programmer in me screams out.

I really think the community is one of the strongest group of programmers I’ve ever seen. I don’t want to belabor this and dwell on the big brain memes, it’s more that they think hard on this stuff and actually push forward, vs just telling each other that web frameworks are rocket science and it’s impossible to do better than what it exists.

Ultimately, Haskell fits like a glove for our domain of program analysis. Beyond that, I’d still be a bit wary. I’m still thirsty for a PL that is essentially OCaml but with a better syntax. But that’s just me.

***runeks*** > I miss early return dearly, and understand why it isn’t a thing (except if you have a MonadZero at hand) but I know it’s a syntactic transformation that won’t make it in.

Early return is mandatory for readability sometimes. I suggest using ExceptT for this: https://www.stackage.org/haddock/lts-21.8/mtl-2.2.2/Control-....

You don’t even have to expose ExceptT in your interface; you can just use it internally for early return and have your function return an Either.

***girvo*** > I’m still thirsty for a PL that is essentially OCaml but with a better syntax. But that’s just me.

Not just you, me too! In fact it’s why I went in deep on Reason when it arrived initially. Shame it never really got traction.

***chubot*** Me three! I came from Python (now with MyPy), learned OCaml and liked some aspects, was intrigued by Reason -- and also sad it seems to be in limbo.

I also miss early returns, and break/continue.

I would like "modern ML" / "Python with sum types" / "Rust with GC" language (indentation/braces doesn't matter to me). Many people seem to agree.

Recently I found TypeScript is kinda fun for this, at least if you're starting from no code, without ecosystem baggage:

https://news.ycombinator.com/item?id=37171801

AFAIK TypeScript's type system can do everything in OCaml -- it's extremely expressive -- but it's dis-similar in that it doesn't use the types to compile to native code. I view that as a downside because JITs are unpredictable and also huge.

It has early return/break. The syntax is pretty conventional, with the usual JS weirdness that everyone has to know.

***consilient*** > AFAIK TypeScript's type system can do everything in OCaml -- it's extremely expressive

Extremely expressive and unsound. And not just in a trivial "escape hatches exist but you should never use them" way - until you've been burned enough it's not at all obvious which operations are unsafe, and there are a lot of them.

***chubot*** I think if you're writing code from scratch, this doesn't really apply -- I'm talking about prototyping language implementations without any libraries at all, sorta like you would do with OCaml from a textbook (e.g. TAPL by Pierce)

(I'm aware of all the terrible experiences people have with TypeScript in the NPM ecosystem. But TypeScript is a big, mature tool and you can use it in more than 1 way.)

I just noticed the 'deno check' command I'm using turns strict mode on by default, so that's good.

https://deno.land/manual@v1.4.1/getting\_started/typescript

All widely used gradual type systems are unsound because they have to interoperate with untyped code, and the dynamic checks to make it sound are too expensive.

But code written from scratch doesn't have that issue. I'd be interested in a counterexample -- is there a code snippet that passes the strict mode of the compiler, and doesn't interoperate with untyped code, but produces an unexpected runtime error?

I guess by "unexpected" I mean that, at runtime, an operation is performed on a value which is not allowed, and the program fails

---

I googled and found this -- https://effectivetypescript.com/2021/05/06/unsoundness/ -- not sure I agree with some points, e.g. array out of bounds isn't unsoundness! The OPERATION is legal, but the data isn't, which isn't something that any type system will tell you.

Similar to divide by zero -- a runtime error does not imply unsoundness.

Also, casts can produce unexpected runtime errors by definition -- that's why they are casts, and you have to opt in! Bad article.

---

I think these are better examples: https://news.ycombinator.com/item?id=15659657

I believe Java has some of those too. Covariance / contravariance is a common source of unsoundness, but definitely not a dealbreaker for me

***consilient*** > But code written from scratch doesn't have that issue. I'd be interested in a counterexample -- is there a code snippet that passes the strict mode of the compiler, and doesn't interoperate with untyped code, but produces an unexpected runtime error?

You'd think so, right? But no, typescript is deliberately unsound in ways that have nothing to do with gradual typing. Here are a few examples.

Signatures written in method syntax are bivariant, which is not correct

interface Unsound {

f(x: number | string): number

}

interface Unsound2 {

f(x: number): number

}

const a: Unsound2 = { f: (x: number) => x }

const b: Unsound = a

const c: number = b.f("not a number")

Type predicate results survive mutation

const hasA = (x: object): x is { a: unknown } => "a" in x

const deleteA = (x: { a: unknown }) => {

delete x.a

}

const unsound = (x: object) => {

if (hasA(x)) {

deleteA(x)

return x.a

} else {

return "no a"

}

}

Many stdlib types are incorrect. JSON stuff is particularly bad: JSON.parse and Body.json() both return `any`.

You can spread things that aren't objects

const unsound = <X,Y>(x: X, y: Y): X & Y => ({...x, ...y})

const bad: never = unsound(5, 4)

(And even for objects, `X & Y` is not the correct type when you have overlapping keys)

Anything with optional fields can be widened incorrectly

const unsound = <T extends { x: number }>(t: T): { x: number, y?: number } => t

const bad: number | undefined = unsound({ x: 5, y: "not a number" }).y

Assignment doesn't handle `readonly` properly

interface Readonly {

readonly x: number

}

interface Mutable {

x: number

}

const a: Readonly = Object.freeze({x: 5 })

const b: Mutable = a

b.x = 4

> The OPERATION is legal, but the data isn't, which isn't something that any type system will tell you.

There are some that will, though unfortunately none that are really production-ready yet.

***chubot*** Great examples, thanks!! I typed them all into the TypeScript playground.

I agree this is weird, and seems to follow from TypeScript's heritage as "trying to describe whatever dynamic JS does"

I mean that's probably why I didn't use it for >10 years (in addition to its JS heritage). But I did find that there is an interesting subset, at least for playing around.

I think the JSON.parse() issue is fundamental -- it's not clear what they could have done better, and static languages don't really do better. There is a fundamental problem there -- type systems are interior to a process, while data is exterior (https://www.oilshell.org/blog/2023/06/ysh-design.html)

I'm going to read this static TypeScript paper -- https://www.microsoft.com/en-us/research/publication/static-... Hopefully that's a sound subset :)

***consilient*** > I think the JSON.parse() issue is fundamental -- it's not clear what they could have done better, and static languages don't really do better.

The best solution, IMO, is to give up on "no type-directed emit" (which harms the language in lots of other ways as well) and derive appropriate parsers at compile-time. Parsing malformed data should fail immediately, not just when you try to use the broken parts. This is a solved problem in C#, C++, Haskell, and no doubt many other languages.

Failing that, it should return an appropriate `JSON` type. Something along the lines of

type Field = string | number | boolean | null | JSON

type JSON = {[key in string]?: Field } | Field[]

***brightsize*** Have you had a look at Coconut? I don't know if it'll push all your buttons but whenever I hear someone who's reasonably content with Python but wants more FP goodies I always think of it. https://github.com/evhub/coconut . It's basically a superset of Python3 that transpiles into Python3 and is compatible with MyPy. I don't think I'd code Python w/o it ever again assuming I had the choice. The biggest negative for me is that there's no IDE support for the language last I looked, though of course you can work with the transpiler output (plain Python) in your favorite Python IDE. It might be fun to play around with, I know that I really enjoyed it but then I got spoiled by the language+tooling of Scala3, but if you don't have that option ...***chubot*** I actually want the imperative Python style, but with sum types. So it's more like "Rust with GC" I suppose.

I used to write in a functional style, and then I wrote Python for decades, and my brain flipped. Now I like imperative code :) I guess it's all the usual things about liking break / continue / early return, local mutation, flat code rather than nested code, etc.

***zogrodea*** I wanted to be a fan of TypeScript and get to use it daily on my job, but actual experience made me dislike the language. I think you already know the pain of external libraries (Express in my case) since you mentioned the ecosystem baggage, and the lack of pattern matching is another big minus for me.***belmont\_sup*** Wish I had the capacity of time and energy to create this dream language of mine too.

Roc lang seems to be building up to what I desire. But we shall see. At the end of the day, picking one of the mainstream runtimes ids the safest bet. F# if we want to enjoy some fun but stay pragmatic.

***GregarianChild*** > PL that is essentially OCaml but with a better syntax.

Scala 3!

Python-ish syntax, much larger library ecosystem (due to JVM) than either Haskell or Ocaml. Better integration of OO and FP than Ocaml. So similar to Ocaml that idiomatic Ocaml has a simple transliteration.

***d\_burfoot*** As someone who uses a lot of "core" Java (ie not the messy ecosystem), and gets a lot of really complex stuff done with it, I read these articles about high-tech language features like algebraic data types and ultra-strict typing, and I think, what are these people actually doing? The vast majority of software engineering consists of simple operations that move data from one place to another - from a DB to a JSON file, from a REST endpoint to a browser screen. Is all this machinery really helping? Are you sure?***yayitswei*** The Blub Paradox is relevant here: http://www.paulgraham.com/avg.html

It's hard to know what you're missing if you haven't tried it. If you see patterns moving data around, it's nice to abstract those out. And higher-level languages give you more powerful abstractions.

***zzo38computer*** I have used many different programming languages (including BASIC, C, Python, JavaScript, Java, Perl, PHP, Haskell, Lisp, and others), even with powerful abstractions (which I can understand how helpful they are), and I still think C is better, and mostly use C. There are improvements which could be made (and I have some ideas of such), but most of the stuff I have seen is usually just worse instead. Some GNU features are good though, and can be used in C, so I often use them.***rowanG077*** If you seriously think C is THE best programming language you should simply be banned from writing code. Seriously. C is completely memory unsafe, all languages which are memory safe surpass it by default.***zzo38computer*** I do not think C (or any other programming language) is the best programming language. However, I think that the memory safety often gets in the way, and that even if it is helpful (which it often is), memory safety is not the only important or helpful feature anyways.***sullyj3*** Possibly you were using "algebraic data types" as a general stand in for fancy type system stuff, but algebraic data types are actually one of the least fancy haskell features. They're much more straightforward than the name might lead you to believe. I think there's a broad consensus that new statically typed languages ought to have them, eg they've been adopted by rust and swift. I find languages that don't support them very irritating to use.***root\_axis*** > The vast majority of software engineering consists of simple operations that move data from one place to another

All computing "consists of simple operations that move data from one place to another", that's essentially the foundation of computer science.

It seems you're trying to say "most software isn't that complicated", which obviously isn't true, in fact, the opposite is true: most software is a complicated mess.

***LinXitoW*** Obviously ADTs are probably just a standin, but I think they are a feature absolutely every language ever should have. They are also not very expensive, esp. runtime wise they're (afaik) equivalent to the clunkier solutions (like classic enums).

Have you never had a thing that could be EXACTLY one of two things, ever? And you wanted the compiler to make sure that, anywhere you used that thing, you had to take care of BOTH of those possibilities? That's one of the main uses of ADTs.

***grumpyprole*** If Java is so great at solving these "simple" problems, then why do hugely complex frameworks like Spring exist? The language features of Haskell that you mention can describe your "simple operations" symbolically, you then just need to write a few different interpreters, one for real services, one for testing etc. No dependency injection, aspect-oriented programming or AbstractSingletonProxyFactoryBeans needed. You might feel that the complexity has just moved, but I'd much rather invest my time in solutions that are not ad-hoc.***gmfawcett*** Strawman? Spring was not designed for solving simple problems.***grumpyprole*** Maybe. I've seen few Java apps that don't use Spring or something similar. The parent described a simple domain, arguing it is the common case, but nonfunctional requirements like testing typically make the problem more complex. So I dispute that Java is good enough for the common case and that nothing from Haskell would be beneficial. Incidentally, Java's leadership agrees.***consilient*** > Is all this machinery really helping? Are you sure?

Absolutely: ADTs and general-purpose crud stuff are a perfect fit. It's the delicate numeric stuff where typed functional languages are at their least helpful (though I think still somewhat better than imperative ones, except at the ultra-high performance end).

***freilanzer*** Really complex stuff like moving data from one place to another? That's trivial.***tome*** > gets a lot of really complex stuff done with it ... The vast majority of software engineering consists of simple operations that move data from one place to another - from a DB to a JSON file, from a REST endpoint to a browser screen

Is the complex stuff you get done with Java the same as the "majority of software engineering" you describe? I'm having trouble reconciling those two claims ...

***blueberry87*** What's your workflow for adding a new feature to something? In OCaml, with it's "high-tech language features like algebraic data types and ultra-strict typing", my entire workflow is: - Modify the types to add the new thing. - Fill in all the pattern matching cases that the compiler tells me I need to fill in. - Maybe write a little bit extra logic. And then i'm done, with a solution I know is type safe, will never crash unless i've explicitly let it, etc. How does this help with moving data? Well, wouldn't it be handy if the compiler could stop your program from crashing when something goes wrong, or you could have the compiler track where you've safely and unsafely done operations, or a million other things. Maybe this is too strong a statement, but if you can't see the advantage of that, I don't want to work with code you've written, because you're not doing everything you can to make it as safe as possible.***Tyr42*** When writing a compiler it helps for sure.

Though most of my time is protobuf in Java, I still wouldn't mind having an ADT or two for when I got lists of things and the things aren't exactly uniform but I don't want to make a type hierarchy.

***mbo*** Doesn't `sealed trait ... permits ...` and `record` satisfy that these days for Java? That's what our proto2 `oneof`s generate into.***Tyr42*** > I don't want to make a type hierarchy

Yeah I could emulate it, but it's much more verbose, and the signal to noise ratio is bad. Ex, a four line Haskell ADT could be 40 to 100 lines of java, and you gotta think things through a bit more. Instead I might have two fields which are Optional and specify an invariant in the comments.

***chpatrick*** I recently went to one of the largest Haskell meetups in Europe and pretty much no one used Haskell any more (including some formerly core people), it was almost just a social gathering.

I think in 2023 many of the things that made Haskell appealing compared to other languages before have been widely adopted, while the developer experience and ecosystem for Haskell is as bad as it was. I wouldn't use it for a new project outside of some specific areas.

***nh2*** I went to the same meetup (ZuriHac), and arrived at the opposite conclusion.

I gave a lightning talk there on how the Haskell job market has been growing steadily since 2008 [1] [2].

The GHC bug tracker is full of new people filing bugs from production environments.

Consultancy blogs such as [3] regularly show industry-sponsored improvements to GHC, which was much more infrequent 10 years ago.

A this year's ZuriHac, around 50% of attendees were new to Haskell / had never visited ZuriHac before (this was an audience question).

In the past, there were a few well-known companies that used Haskell, in specific niches. Today, the big niches are diminished, and there are more companies that use it in more niches.

> the developer experience and ecosystem for Haskell is as bad as it was

The developer experience improved significantly over the last years.

Today, you can get a good quality IDE environment with VSCode and Haskell-Language-Server that works in both simple and complex environments, and includes all the features you'd expect (completions, immediate type error checking, scoped renames, go-to-definition, find-all-references, call hierarchy, docs-on-hover).

[1] https://news.ycombinator.com/item?id=36742311

[2] https://github.com/nh2/haskell-jobs-statistics

[3] https://well-typed.com/blog/

***chpatrick*** > The developer experience improved significantly over the last years.

> Today, you can get a good quality IDE environment with VSCode and Haskell-Language-Server that works in both simple and complex environments, and includes all the features you'd expect (completions, immediate type error checking, scoped renames, go-to-definition, find-all-references, call hierarchy, docs-on-hover).

In 2023 Haskell has indeed kind of reached 2003 levels of IDE support but you can forget about a working debugger, practical compile times or you know, stack traces.

***chpatrick*** Simon Peyton-Jones, the very inventor of the language is now working on Tim Sweeney's Metaverse-themed prolog bullshit. There's no energy left in Haskell.

Haskell has fallen between the cracks as neither a super efficient compiled language or a practical interpreted one. It's just a pain in the ass.

***csb6*** SPJ still allocates part of his working time to Haskell [0]. Also he did not invent Haskell.

[0] https://discourse.haskell.org/t/an-epic-future-for-spj/3573

***Capricorn2481*** What were some things people were using?***chpatrick*** Rust, TypeScript, C++.***friend\_and\_foe*** I just started learning Haskell last year (at my own pace on my own time) and from the minute I wrote a recursive function I understood how freeing and smooth writing in Haskell was going to feel. Once I got used to the new concepts this was going to be like butter...

The two problems I have with it are the archaic standard library and the tooling. Because everything is atomic and strongly typed, you basically have to rote memorize the standard library before you can use any of it. The tooling is just clunky, I can't come up with a better term. The author is right IMO. I haven't been writing it long enough to get the rug pulled out from under me with regard to breaking changes and that, I guess we will see how that goes.

But the language itself... It's like finding this magical thing. I wish Haskell had better tooling and at least more approachable documentation for the prelude.

***regularfry*** Hm. Not really convinced by either Ruby counter-example. The first one's ok, ish, but doesn't take advantage of the fact you can pass a block to `zip` so you don't need the `map` call.

The second one's just wrong. You wouldn't use `flat\_map` for that if you didn't want indentation, you'd use `Enumerator#product`:

def all\_flavors

flavors = [:vanilla, :chocolate, :strawberry]

containers = [:cone, :cup]

toppings = [:sprinkles, :nuts, :fudge]

flavors.product(containers, toppings).map { |flavor, container, topping|

["a #{container} of #{flavor} ice cream with #{topping} on top"]

}

end

While that's not quite doing the same as what `pure` and `<>` do in the Haskell example, the complaint was in terms of visual layout and in that respect it's very similar.***Tainnor*** The point is that Haskell's do notation works for every monad, whereas "product" in Ruby works for just a cartesian product of sets, and not any other use of monads (e.g. generating random values, async code, operations that might return errors, IO operations, and so on).***regularfry*** The point is more specific than that: the claim here against this example is that Haskell's `do` notation uniquely lets you avoid indentation depth. The thing is, none of those other examples cause indentation depth problems in other languages because they don't force you through type system hoops to get useful work done.

Besides, you can (ab)use Enumerator for all sorts of monadic things if needs be and end up with something quite terse. A random number generator is trivial, for instance.

I'm not saying that `do` notation isn't a neat trick, it just tends to be a trick other languages don't need. Its generality is both a blessing and a curse.

***Tainnor*** > The thing is, none of those other examples cause indentation depth problems in other languages

They do, I've seen it happen before. Callback hell was a thing in JavaScript, before Promises happened (and later async/await). And if you've ever written in a functional programming style in a language that doesn't have a first class abstraction for monadic code, then you might have seen cascades of flatMap and map.

FWIW, do notation is not the only option, Scala has for comprehensions that accomplish the same goal.

***twic*** "Haskell is best at solving problems that Haskell invented and other languages do not have." -- Jon Harrop***asplake*** To me the ‘product’ in the Ruby example above is helpfully explicit and could be changed for something else.***paldepind2*** You're absolutely right. That is particular example is bad because you don't actually need the power of flapMap/bind. But, if the available containers depended on the flavor and the available toppings depended on the flavor and/or container, then you would need flapMap/bind and do-notation would pull its weight.***lykahb*** I've maintained a Haskell library for databases for over ten years. Here is my take:

Haskell is a complex and a flexible language. It pushes you toward correctness, but in other dimensions is less opinionated than other languages. If you choose to stay away from the low-level Template Haskell (the types for it are updated often) and the bleeding edge type system tricks, Haskell would be quite stable.

The idea to use only the simple parts of Haskell is great - but what is simple is a rather subjective judgment. Oftentimes the technical choices would be made before the the impact on dev ux and effort to maintain becomes clear. Luckily, doing large refactoring in a complex Haskell project is safe, and improving over the initial choices is easier than in most other languages.

***hardwaresofton*** Site got HN hugged (probably):

https://web.archive.org/web/20230824095246/https://journal.i...

***yakshaving\_jgt*** It's strange that under any article like this, there's always commentary along the lines of "Hmm, yes, indeed $LANG is bad. What shall we all migrate to instead?"

Reminder that this post represents one person's opinion.

Haskell is still just fine as a programming language for getting actual work done.

***lihaoyi*** If you like Haskell but want something else, you really should consider Scala.

It's not the same. But it has many of the same niceties around the rich type system, but with generally good tooling, the amazingly rich JVM ecosystem (tooling, libraries, learning materials), and a somewhat more pragmatic bent to it.

Scala has a bad rep, justifiably so, due to a lot of its problems in the past: community, libraries, tools, etc. But many of those past problems are much better today. Scala today is a much better platform than it was at peak-hype circa 2015, despite some ongoing warts. Nowhere near perfect, but pretty good overall

I see people in the comments looking for a "better OCaml" or a "industrial Haskell", those folks should definitey try Scala

***reidrac*** Scala has mostly all the issues the post is discussing. Perhaps once you're on Scala 3 you're going to enjoy stability, but your dependencies most likely won't give you that.

And the JVM, and compilation times, and poor Scala 3 support on editors / IDEs.

I prefer the Haskell tooling TBH.

***yw3410*** The Scala community is incredibly committed to stability; every open source maintainer I know of in that space checks for binary compatibility when releasing as well as cross-compiling for multiple targets (Node, Web, Native) and versions.***valzam*** The problem with scala is that the core devs seem to be working against what the community wants. At it's core it's still a research platform for many and it shows in the priorities for what gets included in Scala 3. Also while individual open source libraries are generally very stable and high quality the overall ecosystem is very fragmented and there is lots of churn (cats vs zio holy wars etc.).***otter-in-a-suit*** I agree. Scala is actually usable for real-world use cases on a much broader scale, simply by virtue of being on the JVM. I wrapped some of the (unergonimic-because-code-gen'd) AWS libraries in cats-effect just the other day, matter of fact. Some of the native libraries do suffer from this "stylistic neophilia" too, imo, but maybe not to the same degree as Haskell.

Also, Scala 3 made a lot of great improvements, although the tooling and library support arguably regressed w/ IntelliJ (I assume that's temporary) and some of the (on paper) positive changes - especially the whole type class derivation topic - make older articles and books \_very\_ confusing for people just getting into the language. I'd say making that migration happen in larger code bases and companies will continue to be a real-world challenge, but hopefully in the shape of a one-time effort, rather than constant changes.

Also, based on the username, I assume that the parent comment is by Li Haoyi. Funny how small the scala world is - I tend to see the same names (you, Alvin Alexander, Gabriel Volpe, John A De Goes) over and over. :)

***chriswarbo*** Due to Scala's JVM heritage, it takes some discipline in order to have confidence in Scala code (e.g. that it won't throw exceptions, or return nulls, or do weird type casts, or overflow the stack, or have spooky action-at-a-distance, or have race conditions, or do inexhaustive pattern-matches, or not actually accept/return the types in its signature, etc.). This can be a tall order for shops which have a Java background (e.g. my last job used Scala extensively, but most had a Java background and treated Scala like a different syntax for Java; I had written more Haskell and ML, and treated it like an ML with JVM-gotchas). I highly recommend turning on all the scalac -Xlint options, using other linters like WartRemover, and treating the warnings as errors (annotations can silence the rare cases of "I know what I'm doing"; but there had better be a comment with a good justification, if you want it to pass code review!)

Whilst it's nice that there are loads of Java packages to import and use from Scala, it's usually a good idea to encapsulate them in a more "Scala-friendly" wrapper; e.g. to replace exception-throwing with `Try`, `null` with `Option`, etc. in order to maintain confidence in our code. With a little thought, and a sprinkling of Scala features (e.g. `lazy val`, by-name parameters, implicit arguments, etc.) such wrappers can end up being much easier to use than the original, too!

***theCodeStig*** This exactly.

The Scala community caters a lot to JVM/Java inter-op, and to recruitment of Java engineers. In the peak-hype days, Java compatibility, and Scala for Java engineers learning resources were a huge selling point for Scala. Now it is one of its greatest weaknesses holding it back. Even as late as June 2023, the second edition of Functional Programming in Scala is catered toward Java programmers.

Functional purity is an option in Scala (no pun intended), and the org/team must have the collective discipline to write pure code in-order to make it work.

Pure/lazy effect handling is not included, one must use a 3rd party library. There should be one, or a small handful of compiler options which would enforce purity by the compiler. There isn't. The -Xlint options are not easily discoverable. WartRemover is a 3rd party library, which doesn't get enough visibility. Some of the "good stuff" from Typelevel should be absorbed into the standard lib.

The numeric types are not ergonomic, there is no natural number type for example.

I haven't had issues with Haskell tooling, though I also use Nix. On the flip side, I've never heard anyone say "I love SBT". Yes there are alternatives to SBT (Mill for example), but again they suffer from low visibility. Martin Odersky has admitted faults with SBT, and praised Mill; yet, what does the Scala community push... SBT.

***jsbg*** I found Scala to be missing too much of what I liked about Haskell. If I'm gonna lose out on the niceties that are unique to Haskell, then I'd rather go with something more pragmatic, like Kotlin.***paulddraper*** Scala 3 was a massive step forward.

Though for whatever reason it seems that its popularity is declining. [1]

[1] https://twitter.com/jdegoes/status/1656566825356754945

***wavemode*** Because Scala 3, as great as it is, has fractured the community and caused a lot of churn.

Twice, just in my recent job history, have I worked for companies that were using Scala 2 for a long time but are now deciding to develop new projects in Java and/or Kotlin instead.

***kagakuninja*** I think that is less about Scala 3, and more that the companies were not committed to the use of Scala, and would eventually move to some other flavor of the month language, such as Go or Rust.

The transition to Scala 3 at my current job has not been an issue. Scala 3 is mostly the same language, with some nice new features which are optional. Our old projects are still on Scala 2, as there isn't a huge benefit from upgrading.

The main downside in upgrading was library support, but we are now 2+ years since the release of Scala 3, the issue is mostly solved unless you depend on unmaintained libraries.

***chriswarbo*** Library compatibility was preventing us upgrading from 2.12 -> 2.13; Scala 3 wasn't even on our horizon :(***kagakuninja*** By contrast, over the last 8 years I've migrated a number of small micro services up from every version of Scala since 2.10 with no problems, maybe some minor dependency hell with some of them. I've even upgraded a simple project to Scala 3 with no code changes.

It depends on what libraries you use. The big hurdles tend to involve macros and Spark.

***urthor*** I find the issue is most of the niceties are also built into Rust.

Scala's perfectly good though, there's very little to complain about.

***paulddraper*** JVM debugging+monitoring is leagues better***foderking*** ...or f#, which is actually the "better ocaml"***beanjuiceII*** "I also… don't really want to deal with them on a day-to-day basis. My personal experience has been that very often these sort-of-experimental approaches, while solving some issues, tend to cause many more issues than is apparent at first."

This one really hits home for me

***kstenerud*** It's a similar issue with frameworks in imperative languages. Everything's fine until one day you find that you have to step outside of its envisioned bounds in some way. Then the sadness begins.***lcedp*** The page appeares to be hugged to death. Google cache: https://webcache.googleusercontent.com/search?q=cache:uWo0Ni...***rowanG077*** Interesting. I use Haskell professionally and this article doesn't touch on the most fundamental problem I have with Haskell at all: Function coloring. Basically every monad transformer is different. Just calling basic function from somewhere else can involve lifting. Refactoring is a total pain. Oh you just want to log here but your concrete monad doesn't have a logger? Too bad...

I understand an effect system alleviate this somewhat, so I hope to try this in the future. But holy shit is working with monads annoying.

***consilient*** You should generally be writing code against typeclasses, not a particular monad transformer stack. For example:

fibonacci :: MonadState (Int, Int, Int) m => m Int

fibonacci = do

(prev, prev2, n) <- get

if n > 0 then

put (prev + prev2, prev, n - 1) >> fibonacci

else

return prev2

concreteFib :: ReaderT String (StateT (Int, Int, Int) (ExceptT String Identity)) Int

concreteFib = Fibonacci

***rowanG077*** You misunderstand my problem. Add a logger to that fibonacci function. Potentially EVERY usage site now has to change, maybe even multiple layers. Adding a log in most languages is a local transformation. In Haskell it isn't, it can have codebase wide consequences.***runeks*** That's because most languages allow arbitrary IO in all functions. You can achieve this in Haskell too if you want, by just using IO as your monad everywhere.

But most people don't do that because then it becomes really hard to reason about your code, so spending the extra time propagating a MonadLog constraint up your stack is actually worth it.

***jaspervdj*** If you wanted to log from fibonacci, you would pass a some logger instance down to this function. In Haskell, this could be a record or a typeclass instance. In other languages, it could be an object or a struct. There is no fundamental difference. All the layers above would still have to pass this through; explicitly or implicitly.***rowanG077*** The difference is most language have global state. See rust for example. It's not common to inject a logger, a "world" object for IO or anything for which you need monads in Haskell at all.

You are arguing from a concrete technical standpoint: "But you need to do the same thing in other language if you want to mirror Haskell monads". Sure, you are right. That also completely glosses over the point I'm making. I simply feel like the way haskell does it is unergonomic, it would also be unergonomic to something equivalent in other languages.

I don't know what a good solution would be, maybe a constrained partial type signature? Let the compiler pick the smallest constraint from a larger space of available constraints that fits with the usage and simply let the type checker bubble it up until you actually care to specify it? GHC doesn't support this but it should be possible in theory.

***doyougnu*** Does every usage site have to change? You would alter fibonacci to be:

fibonacci :: (MonadLogger m, MonadState (Int, Int, Int) m) => m Int

fibonacci ...

and now of course all callers must support MonadLogger. But instead of using the MonadLogger (or any mtl constraint directly) you should just be constructing an abstraction boundary with a type class synonym:

class (MonadLogger m, MonadState s m) => MyMonads s m

and now you change fibonacci:

fibonacci :: MyMonads (Int, Int, Int) m => m Int

fibonacci ...

And now if you need to add a monad or add Eq or whatever you just have to change your type class synonym rather than every function. Its not a problem with the language its just programing with modularity in mind, even in the type system.***rowanG077*** I have seen this in the wild. The result often is that every function has a kitchen sink MyMonads constraint of which it only uses a tiny subset. It's death by a thousand cuts. If you make such a class for every monad combination you get insanely large amount of classes. It's simply unworkable. Which is why you get the kitchen sink monad pattern.***runeks*** If you think it's fine that you can log from all functions in other languages, then what's the problem with adding that constraint to all your Haskell functions to allow this?***rowanG077*** The problem is that you should always write your code to be idiomatic in the language. In this case I feel like the Idiomatic Haskell way has serious drawbacks.

For example, It's fine in C to manually allocate/free memory, it's the way you have to write C. It's not fine to do the same thing in Rust. Even though you of course could do that in Rust as well.

***tome*** It's perfectly idiomatic Haskell to annotate all your functions with an effect you believe they should all have.***rowanG077*** The only reason this is idiomatic is because there is no better way. That's the entire point I'm making... Haskell prides itself in writing generic and reusable functions. This then is then thrown out of the window with the kitchen sink monad. Very understandable, because everything else sucks.

That precisely why I think this is a great shortcoming of the language.

***tome*** It's not a shortcoming of the language; it's a shortcoming of the goal! You can't have both the goal of fine-grained effect tracking and the goal of not having to make fine-grained changes when effects change. They're incompatible goals in any language.

The strength of Haskell is that it allows you to achieve the first goal if you want. Most languages don't (pretty much no other language, actually).

***rowanG077*** That's an extremely limited point of view. Just because Haskell allows precise specification does not mean it is impossible to disallow loose specification. In fact that's one of the strongest values of Haskell's type system. For example you can overconstrain your head function

head :: [Int] -> Int

But you can also just leave out the type signature all together to let the type checker figure out the most generic type. You can have your cake and eat it too.

You can even almost do what I want with partial type signatures. Just sprinkle it everywhere inside your constraints. GHC will automatically pick the right constraints. At the call site you actually care for the definition you can not use the partial type signature. The great disadvantage of this is that you now introduce ANY constraints into your type signature and you lose your types as documentation.

But that doesn't have to be

You could have a constraint with something like `UseMonadSubset (...)` which works almost like partial type signature. GHC should infer 0 or more of the monads insde `UseMonadSubset` as the actuall constraint. `

Then you could write something like:

fibonacci :: UseMyMonadSubset m => m Int

fibonacci = -- Uses only MonadState (Int, Int, Int)

-- Type checks because type checker can see fibonacci ONLY uses MonadState

foo :: MonadState (Int, Int, Int) m => Int

foo = fibonacci

bar :: UseMonadSubset m => Int

bar = fibonacci

Which allows for precise specification if you want to and if you don't you let the type checker figure it out. You may even be able to implement this as GHC type checker plugin.***tome*** It's an interesting idea but I can't say I feel that would solve a problem I've ever had. In fact, I always completely annotate top-level definitions with their types. I never want them inferred. And I've never felt it too burdensome to fix up a call stack when adding a new effect. But if you consider that a weakness of Haskell then so be it!***kccqzy*** And what's wrong with the kitchen sink monad pattern? I've certainly used exactly that. And I have no problems with it.***rowanG077*** Because your code is very much overconstrained at that point. For the same reason you don't add a `Num a` constraint to list `head` function. You have now essentially fused your function to your codebase.***kccqzy*** That's not a problem in business logic heavy code. Requirements change and you could use previously unnecessary constraints at any time.***icrbow*** Debug.Trace has a lots of stuff for that. You can even generate charts from that when using eventlog-enabled runtime.***consilient*** If you just want to add logging to existing operations, reinterpret them at the call site. Something like this

newtype LoggedStateT s m a = LoggedStateT (WriterT s (StateT s m) a)

instance (Monoid s, Monad m) => MonadState s (LoggedStateT s m) where

get = LoggedStateT $ do

val <- lift get

tell val

return val

put s = LoggedStateT $ tell s >> lift (put s)

(which is basically an ad hoc effect system)

If on the other hand you want to reproduce the behavior of other languages, throw everything in `MyAppMonad` give it whatever capabilities you need.

***rowanG077*** Which requires sweeping changes... In most other languages it's literally a one liner where you want to log something.***consilient*** It requires changing the places where you instantiate your monad transformer stack, which you should have very few of.***rowanG077*** I don't think having very few is a good scenario. I have written a compiler and had about 10 different stacks. Changing every single one just to be able to add a logger to a single function somewhere is honestly insane.

What I see in the wild is having one huge kitchen sink stack which sucks as well.

***valcron1000*** In the general case, adding IO to any piece of code requires changing all callers. I would argue that it's a feature, not a bug: now `f x` is no longer a value, but an action: calling it twice can result in duplicated logs, for example.

If you need that logging for debugging then you should use `Debug.trace` though.

***rg111*** I want to seriously invest some time into properly learn a functional language.

My goals are simple- write small scripts, solve programming problems in sites like Codewars, Leetcode, Euler Program, etc. And yes, having the "functional enlightenment" or something similar.

Which language should I learn and invest time into? Scala, Clojure, Haskell, OCaml?

***habitue*** Learn Haskell, it's the most elegant, and if you're using it for LeetCode / project euler you can have the experience of polishing a simple elegant program until it shines. It's a very satisfying experience.

Also, Haskell is lazy, which is fun and very different from other languages you'll use.

If you become a Haskell afficionado, it also kind of acts as a secret handshake in interviews. Like, you aren't going to be programming in Haskell here, but "you get it"

***consilient*** Functional programming is not really a single paradigm. In particular lisps and ML style languages are very different and you should try both. Scala is still a C-style language at heart, it's just much further along the borrowing ML features path than most.

I'm not super familiar with the contemporary lisp landscape but scheme is the traditional choice for teaching.

Among MLs I would definitely pick Haskell.

***valenterry*** Haskell. I prefer Scala as a language to do productive things. But if you want to learn a pure functional style (that you can apply in Scala later as well) then Haskell is better becaue it forces you into this style. Few other languages do that.***iainctduncan*** Dissenting vote: Scheme (or Racket).***rg111*** Thanks to everyone who answered.

And I now notice that autocorrect made Project -> Program. :/

I will keep a watch here for more responses.

***yakshaving\_jgt*** Haskell.***GregarianChild*** Among the ML descendants, I suggest Scala 3, because it has essentially all the power that we like in Haskell, (HTKs, good support for ad-hoc polymorphism), but runs on the JVM, a mainstream platform with a vast library ecosystem.***yakshaving\_jgt*** I haven’t used Scala in a long time, and I’m guessing it isn’t as gross now as it was in 2014.

Your point about the vast library ecosystem might be valid. Personally, there has only been one time in several years working with Haskell I’ve wanted to use a library that I couldn’t find an analog to in Haskell. It was WeasyPrint which is a Python thing, and there was no problem to run it from my Haskell program as an external process.

Although thinking about this some more, I probably could have just used Pandoc…

***chubot*** This part is interesting:

A good concrete example here is a compiler project I was involved in where our first implementation had AST nodes which used a type parameter to represent their expression types: in effect, this made it impossible to produce a syntax tree with a type error, because if we attempted this, our compiler itself wouldn't compile. This approach did catch a few bugs as we were first writing the compiler! It also made many optimization passes into labyrinthine messes whenever they didn't strictly adhere to the typing discipline that we wanted: masses of casts and lots of work attempting to appease the compiler for what should have been simple rewrites. In that project, we eventually removed the type parameter from the AST

which also seems to conflict with this:

Using data structures indexed by compiler phase is a good example of a “fancy type-level feature” that I've found remarkably useful in the past.

Both of these sound like the "AST typing problem" - https://news.ycombinator.com/item?id=37114976

which I admit I'm a bit skeptical of, because the problem is type safety, and not the compiler's actual algorithm or actual performance.

But I guess the first one is for syntax trees, and the "trees that grow" paper (linked in the article) is for back end passes? Does that change the problem so much?

I'm not experienced with back end passes for compilers, but I personally don't see the problem of using either a Map<AST, ExtraInfo> or a nullable field.

I just hacked on a toy codebase that had the Expr<void> and Expr<T> type safe solution, and it's interesting. But my first impression is that it causes more allocations and makes the code a bit longer.

---

I guess another way to justify the Map is that it's like math -- a "typing relation" is an association from expr to type, so a map or multi-map seems natural to model it.

***andolanra*** The former is referring to representing an AST using a GADT in order to include the typing discipline of the target language in the host language. For example:

data Term t where

Num :: Integer -> Term Integer

Bool :: Integer -> Term Integer

Add :: Term Integer -> Term Integer -> Term Integer

IsZero :: Term Integer -> Term Bool

IfThenElse :: Term Bool -> Term a -> Term a -> Term a

With this AST, you can express well-typed programs like `Add (Num 2) (Num 3)`, but the Haskell type system will stop if you express an incorrectly-typed program like `Add (Num 2) (Bool False)`.

The "Trees That Grow" paper, on the other hand, is about reusing the same AST but gradually adding more information to the nodes as you progress through the compiler. For example, you might want to start with variable names being raw strings (so that a term corresponding to `lambda x: lambda x: x` looks like `Lam "x" (Lam "x" (Var "x"))`) but eventually replace them with unique symbols so that shadowed names are non-identical (so that under the hood it looks more like `Lam 1 (Lam 2 (Var 2))`, although in practice you'd want to keep the old name around somewhere for debugging.)

One way to accomplish this is to introduce an explicit type-level notion of compiler phases, give your terms a type parameter which corresponds to the phase, and use the phase to choose different representations for the same nodes:

data CompilerPhase = Parsed | Resolved

data Expr (phase :: CompilerPhase)

= Lam (Name phase) (Expr phase)

| App (Expr phase) (Expr phase)

| Var (Name phase)

type family Name (t :: CompilerPhase) :: \*

type instance Name Parsed = String

type instance Name Resolved = Int

Using this example, an `Expr Parsed` will contain variables that are just strings, while an `Expr Resolved` will contain variables that are integers, and you can write a pass `resolve :: Expr Parsed -> Expr Resolved` which just modifies the AST. (This is a toy example: in a real compiler, you'd probably want to create a new type for resolved variables that still keeps a copy of the name around and maybe some location information that points to the place the variable was introduced.)***chubot*** Ah thanks for the answer, funnily enough that's what I recently read here - https://dev.realworldocaml.org/gadts.html

They have a bool / int expression language example. (And funny thing I'm coding up such a type checker and evaluator right now in TypeScript. TypeScript union types seem to be pretty expressive and useful for this problem.)

However I'm STILL confused ... Admittedly I skimmed through the GADT chapter and I didn't follow all of it, but I don't understand why the answer isn't:

1. have a untyped representation that allows (Add (Num 2) (Bool False))

2. write a type checker on that representation

3. The output of the type checker is a new IR, which can only express (Add 2 3) => Int and (Eq 5 (+ 3 2) => Bool

Is the more abstract GADT solution supposed to "save" you some work of writing a type checker by somehow letting you reuse Haskell or OCaml's type system to express that?

That's a little weird to me ... it seems to be confusing the metalanguage and the language being implemented.

It's not surprising to me that this would go awry, because in general those 2 things don't have any relationship (you can implement any kind of language in OCaml or Haskell).

---

Hmm OK I actually DID run into the issue where you need dynamic checks in the evaluator, that should be impossible BECAUSE the previous the type checking phase passed.

i.e. some of the rules you already encoded in the type checker, end up as "assert" in the evaluator.

Hm I will think about that. There is some duplication there, sure. But I still think there's a weird confusion there of the language the compiler is written in, and what the compiler DOES

You're kinda coupling the 2 things together, to avoid a little duplication.

I also wonder how all that is actually implemented and expanded into running code. The GADT syntax seems to get further and further away from something I imagine can be compiled :)

---

edit: Thinking about it even more, although I did run into the dynamic checks issue, I think it's because my type checker only CHECKED, it didn't LOWER the representation. So I think what I originally said was true -- if you want more type safety, you introduce another simple IR. What's the benefit of using GADTs ?

***consilient*** > That's a little weird to me ... it seems to be confusing the metalanguage and the language being implemented.

Well, that's sort of the point. It's perhaps a little less important if you're writing a compiler for a separate programming language, but that's a relatively rare use case. GADTs are mainly used for implementing DSLs to be used elsewhere in the same program.

> Is the more abstract GADT solution supposed to "save" you some work of writing a type checker by somehow letting you reuse Haskell or OCaml's type system to express that?

Yes. Writing your own typechecker and integrating it into an existing compiler pipeline is a ton of work, easy to screw up, and a potential compatibility nightmare.

> I also wonder how all that is actually implemented and expanded into running code. The GADT syntax seems to get further and further away from something I imagine can be compiled :)

GADTs are syntactic sugar for constrained type constructors. So this

data Example a where

Example :: Int -> Example Int

becomes this

data Example a = (a ~ Int) => Example a

which is then compiled to a type coercion in the intermediate language***spopejoy*** Haskell continues to punch above its weight in divisiveness. When was the last time you saw an article hating on F#, OCaml or some lisp get this kind of attention?

Haskell hate has an distinguished history. In the 2000s OCaml was "winning" in FP-for-production-apps land, which for no good reason resulted in an outpouring of vitriol for Haskell.

Steve Yegge's joke article comes right at the inflection where Haskell "grows up" and starts offering compelling advantages for production apps, but nobody knows it yet.

The 2010s see real adoption and fervor but no drop in click-grabbing haskell hate. If anything it increases, although the OCaml community buries the hatchet.

So it almost warms my heart to see this still getting attention in the 2020s. Switching to Haskell a decade ago changed my life for the better in every way, so I'd of course rather see Haskell love climbing the charts. But this trend instead supports the thesis that Haskell has "stayed weird", which is probably a good thing.

***IceDane*** I could have written this article.

I discovered it around the same time, and it was my go-to language for years and years. I had a brief stint where I wrote it professionally too, but nothing serious.

Eventually I really really got going in the industry and had to use other technologies, namely TypeScript and others. I became pretty fond of those, after getting over the initial hurdles, and didn't really spend time using Haskell at all.

After a while away, I went back to it. One time for a job interview, too. It just .. really lost its shine for me. It just all felt like such academic, ivory tower circlejerk. The ecosystem is still relatively small, the packages in it not always great and almost always have only a single or a couple of maintainers. There are still things missing from the Haskell ecosystem that people wanted over a decade ago. Hell, things like Streaming are still barely a solved problem. There are still popping up new libraries to solve it, and each does more arcane things than the previous in some attempt at getting streaming to be nice, type-safe while also not leak memory and what have you. Most of these end up leaking GHC-isms into their code.

For the most part, I've just taken what FP or Haskell has taught me to other language. Use the type system(but not too heavily) to help maintain your invariants at compile-time. Write small, pure functions as much as you can and compose them to build more complicated functionality.

I've still sort of kept on top of how haskell has been moving, and it seems to me that a lot of Haskell shops have dropped it as well. I think many may have moved to Rust. I don't see myself ever using it for anything serious again. I would much rather use Rust and get much better DX and performance while still being able to write mostly functional code.

***porcoda*** I had a pretty similar experience: spent a decade (2007-2017) working professionally in Haskell and just got completely fed up with the state of the language, ecosystem, and community. I migrated most of my new work to Ocaml and haven’t looked back.

For me I think the failure of the Haskell Prime effort to establish a successor standard to Haskell 98 was a big factor: the language and ecosystem became more chaotic and inconsistent over time as people drifted away from any hint of a common standard. Add to that the changes in the community when there were big changes in the cohort of “leaders” who had driven things from the 90s until the early 2010s (eg, when both Simons changed their roles). The folks who took their place haven’t done good things for the language in my opinion.

***icrbow*** For the reference, 2017 was the year of GHC 8.0. Since your decision to never look back there were a lot of good things.

The standard didn't come out because of some failure to make it. It was mostly the lack of interest that killed it. I wouldn't be betting that some alternative universe where Haskell Prime pulled through had a noticeable increase of adoption because of this.

Looking at proposals, arguments "from standard" don't tend to generate enough support. What wins hearts is alleviating someone's pain without taking disproportionate externalities.

***porcoda*** I’ve paid attention to the language since 2017 (it’s kind of impossible not to if you work in the PL research field). I just consider it dead for my own work - that’s the sense in which I haven’t looked back.***solidsnack9000*** The author does a good job summarizing ways in which thinking through your Haskell programs is easier than in many other languages. There is a certain straightforwardness to it that, once grasped, is eye opening, appealing and fun.

Haskell opened the door for me to much more rigorous reasoning about correctness.

***andrewstuart*** TLDR: the Haskell programmer who gives a shit has stopped giving a shit.

\* http://steve-yegge.blogspot.com/2010/12/haskell-researchers-...

***neuromanser*** My two decades of professional programming suggest that it's way easier to find a $fringe-lang programmer who stopped giving a shit than a mainstream programmer who's ever given one.

"The worldwide programming community met up over beers today to celebrate their unprecedented discovery of an industry programmer who gives a shit."

I'm a huge Yegge Stan btw, his post is funny AF (as always), don't read this as a knockdown!

Edit: tweaks, grammar, autocomplete fails.

***hackandthink*** Seth Briars could be me.

really funny, thanks

***m1keil*** thank you for that xD***anacrolix*** Completely agree with the author. Haskell is a great language but the ecosystem sux. I have a small project that breaks every time I update stack. It really lacks some high quality, bomb proof libraries, like for http for example. Cross compilation is next to impossible, stack also recently broke for using Docker for that.

Cabal is a mess, as is its weird AF format.

cargo is a dream by comparison. I am not happy with some things about Rust either but by comparison it has amazing tooling, and some incredible solid libraries. It also helps that you don't have to be a wizard to get extremely good performance.

***tempodox*** > the constant changes that [...] cause regular breakages

Wow, for a language as mature as Haskell I find that surprising. I never really got into Haskell for other reasons, but this one warns me to stay away for the foreseeable future.

***jeremyjh*** I Haskelled quite a lot from 2014 to 2017 on a futile side project, and developed/extracted and maintained a few libraries on Hackage and a couple that were included on Stackage. In my experience most of the breakage comes from dependencies, not the language. The biggest shift in my time was the Functor-Monad-Applicative Proposal and I don't think that broke anything for me. Technically it was not even a change in the language but in core type-classes distributed with GHC and used in every Haskell program.

As far as I know, Haskell 98 still compiles most of the same programs it did 25 years ago. Some extensions that are commonly used have introduced breaking changes I believe, but I don't remember the details. But I gave up maintaining my projects because 1. I wasn't using them (because I moved away from Haskell, for mostly the reasons in the OP which I feel like I could have written myself almost) and 2. my dependencies kept breaking them.

***habitue*** Haskell's motto is "avoid success at all costs", and part of this is making these kinds of backwards incompatible changes to clean things up.***yakshaving\_jgt*** To clarify, it's not "avoid success (at all costs)". It's "avoid: success at all costs".***pmarreck*** For anyone new-language-curious, Idris is a very interesting project that is inspired by Haskell but which seems to have less design drawbacks***jokethrowaway*** I tried to make Haskell work as a product language for a long time - and, until Rust was around, there was really nothing better than Haskell in terms of guarantees.

Haskell is a research language: it's supposed to change and experiment with new things; the permanent in-flux state is by design albeit I wish we crystallised more of the good PRAGMAs as default.

***blagund*** My notes on using Haskell: mostly agree with the article, but I came to embrace rolling my own tooling, some abstraction on top of cabal. I use my abstractions, not what whoever would force on me. Use Haskell as your typed lisp.. with all the pros and cons.

On production use... Don't even get me started. Will work, but need shedding blood.

***massysett*** Typed lisp? Unlike Lisp, Haskell has enormously complex syntax, so generating code in it is a huge hassle.

Template Haskell is something to use only when absolutely necessary, while Lispers write macros without a second thought.

I considered your method of writing my own tooling, but have had a vague feeling that Lisp works better for this frame of mind.

***spopejoy*** Haskell has almost no syntax, even `$` is a userland function. Are you arguing against the feature of allowing ad-hoc infix operators in userland?***massysett*** "almost no syntax" is, I suppose, a matter of opinion. I wouldn't consider these sections from the Haskell 2010 Report to have "almost no syntax."

https://www.haskell.org/onlinereport/haskell2010/haskellch3....

https://www.haskell.org/onlinereport/haskell2010/haskellch4....

https://www.haskell.org/onlinereport/haskell2010/haskellch5....

Dealing with this in Template Haskell is unwieldy at best.

***kagevf*** > the experience of code refactors via algebraic manipulation is still possible in other languages, especially in non-pure functional languages like Scheme or SML

Wouldn't "code refactors via algebraic manipulation" require static types? How would this work in Scheme?

***mrkeen*** I think the requirement would be purity not static types, but I agree with your overall objection.

(Then again, you need static types if you want to have pure functions, IO functions, and not mix them up)

***kagevf*** > I think the requirement would be purity

Well, yeah, I think if I had instead said "if not static typing, then how would it be possible?" ... I had no idea how it could be done, but with something like purity - by which we mean deterministic functions, right? - I can at least start to see how it would be possible with Scheme.

***agentultra*** The schism created by the Simple Haskell folks in the community is a rather unfortunate one.

For context, Haskell used to pride itself on being a language where researchers could experiment with industry users on an industrial-grade compiler without interfering with one another. Industry users get new features sooner, researchers get feedback faster, etc. No long revisions of the standards and waiting for implementations to catch up. The evolution of language extensions, specifications, and the standard base libraries have a long history of success with this approach.

However, in the last few years, a group in the Haskell community has felt like they had to push back against the work of these researchers and "protect" the language from adopting them. You end up with folks like the author who feel exhausted by having to argue against adopting these features in code bases they're working on and you have the researchers who also feel exhausted getting constant push back on their ideas and hard work. As this schism has developed it has left people feeling exhausted on both sides where there was once collaboration and community.

Write a style guide. This isn't a problem that's unique to Haskell. You get it in C++ and even Javascript too. If using the entire language is not feasible for your project then state it clearly in a guideline. At the very least it requires contributors to seriously consider their reasons for going against the guide and forces them to justify their changes. The nice way Haskell has done it, from my perspective, is that you basically don't pay any cost for not using those features (caveat being Linear Haskell, but they made it a goal to minimize the impact and I think the cost has been amortized by performance gains in recent versions of GHC at least).

Towing the line against progress in the language seems counter-productive to me. I think there are plenty of language extensions in Haskell to work around the lack of expressiveness in the type system that wouldn't be necessary if the language had dependent types. Learning how to use those extensions and which ones work well together is a whole art that could be greatly simplified by a more expressive type system.

To be fair, I almost never use dependently typed Haskell. And I rarely reach for type level programming... until I can't; usually when I'm writing library code that needs to do stuff with user types... even then, quite rare in practice.

If you sympathize with the author then adopting Haskell may not work for you but I would still consider that there are many technical reasons to use it on a project.

Also, I would also advise folks to avoid telling people they should learn Haskell in order to become better programmers. It's true you have to learn a lot more to go from Java -> Haskell than you would from Java -> C#. However there are practical reasons to be using Haskell that aren't about self-improvement:

- GHC is a battle-tested compiler with decades of industry use that produces some excellent code, has an excellent manual, and has an active community of contributors that are always improving it and making regular releases.

- The concurrency story in Haskell is hard to beat: you want immutability-by-default and STM on green threads. It's excellent.

- The type system is a feature: inference, typed holes, type classes... if you don't have a ton of domain expertise you can still arrive at solutions to complex problems using algebraic reasoning. Clash, Crucible... there are many projects that benefit from having a good industrial-grade compiler that also has an expressive type system.

However I have been using Haskell professionally for a number of years, I maintain a couple of libraries, and I stream myself working on non-trivial Haskell projects for fun and there certainly are drawbacks that I would call out:

- On projects that get into 5000+ modules, build times become difficult to manage and require extensive tooling to stay productive. Haskell does a lot more work than many other language compilers and you have to avoid a lot of features that might be convenient in order to keep build times under control

- Modules are fine but type class constraints can "leak" outside of your API boundaries which can lead to a fair amount of coupling if you're not disciplined about how you approach the design of your type classes.. not really a thing you'll be worried about off the bat though, more of a nitpick

- If you interact with SaaS services chances are you will have to write your own client libraries. For things like amazon AWS you're covered but there are a lot more out there and people don't publish SDK's in Haskell.

***1letterunixname*** "Worse is better" phenomenon.***steno132*** If you want to get things done, don't use Haskell.

You'll be fighting against the type system constantly. You'll be asking questions on Stack Overflow only to get no response. You'll be rewriting software that's in other language's standard library.

Productive programmers don't use Haskell.

***friend\_and\_foe*** You're not wrong, but the thing to know about the type system is once you get it it's like this amazing thing. You start to think about your program in terms of the data it manipulates, you'll find yourself just without effort writing the code that does exactly what you intended, it's pretty wonderful. But on the way there it is painful.***antod*** The bathed in a suffusion of blue phase. (or should that be purple?)***theCodeStig*** The compiler/type system is your friendly assistant. Learn to use it, and it's your greatest asset.***yakshaving\_jgt*** This is nonsense.***greydius*** Anyone hiring Haskell devs?***icrbow*** Yes. The amount of job postings raises year to year.***xedrac*** As a Haskell developer, this was hilarious! Thanks for the laughs.***keyle*** I'm not sure I follow, could you expand for non-Haskell developers?***xedrac*** Oh wow, when I originally clicked on the link, it took me here:

http://steve-yegge.blogspot.com/2010/12/haskell-researchers-...

I see why you were confused.

***tuukkah*** That submission is (now?) here: https://news.ycombinator.com/item?id=37247754***Pannoniae*** Basically, the author's criticism is that the language is too powerful, too expressive, people try very abstract things, tooling is bad and no one cares about the language.

There is something sinister in this - first, in the author's lamentations about bad tooling. Other languages require linters, formatters, static analysers, etc. because the language's built-in features and type system are sub-par. In Haskell, that's not true, so the lack of "sophisticated" tooling is way less painful.

Second, this is similar to the "equalise things by dragging everyone down to the same level"-type of thinking. It "makes maintenance a chore" because the language allows you to be creative and shoot yourself in the foot! Sure, you can have languages like Go where the language is simple and it's hard to write "unreadable" code, but that comes at a cost of being very verbose, and not allowing for an opportunity to write really good code. Basically, enforced mediocrity. "Hard to maintain" is also used as an euphemism for "this is too complex and I can't be bothered to try to understand it" sadly. I find that quite ironic in the face of all the needless complexity in the software industry (just look at k8s, terraform plugins or the web development stack)... I know way too many programmers like that, who argue endlessly about what is the "most readable", which in practice just means to write the most straightforward, laziest, zero optimisation code, and giving themselves an excuse for that. Even if a less-capable language hinders development (either by making things non-verifiable like no proper optional/nullable handling or simply by making the code more verbose and/or less readable) their advocates justify it on the basis of being standard or "best practice". I think Haskell and similar languages intimidate similar people because they can't shrug and say "well the language can't do that" to justify their laziness, but would have to actually make a reason up to why they don't want to write sound, safe and performant programs.

Third, about backwards compatibility.... this is a "damned if you do, damned if you don't" situation. On one hand, you have languages like C++ and JavaScript which are really backwards compatible, and have really sizable drawbacks because of it. Sure, your code will compile in 5 years most likely without changes, but the language as a whole suffers because of it. There should be a place for languages which are ambitious and aren't afraid to break things. Backwards compatibility is simply a design aspect, not some holy commandment you must adhere to all times.

***mathisfun123*** > Second, this is similar to the "equalise things by dragging everyone down to the same level"-type of thinking.

> and not allowing for an opportunity to write really good code. Basically, enforced mediocrity.

> straightforward, laziest, zero optimisation code, and giving themselves an excuse for that.

> justify their laziness,

Jesus christ you have so much disdain for such a large set of people. Have you ever considered that you might have a chip on your shoulder about Haskell adoption?

> There should be a place for languages which are ambitious and aren't afraid to break things.

There is - it's research and hobby, not production. Like literally "not afraid to break things" is the antithesis of an industrial language.

Sometimes I wonder if some language communities really are cults. It's the same with Lisp - replace "type system" with "homoiconic" and you have the exact same mantras about ultimate supremacy and "intellectual" superiority. But the proof of the pudding is in the eating and across probably 10 computing devices in my home, not a single one has any Haskell or Lisp programs running on it. Lots of Java, C, C++, Python, Rust, Bash, JavaScript, even Perl (probably) but not a single binary compiled from Haskell or Lisp interpreter to be found. To any reasonable person, that should be a strong signal about the so called "power" and value of their sacred cow.

Put another way: the word language in the concept of programming language isn't accidental and learning a niche programming language is comparable to learning a niche human language - have fun, expand your horizons, but don't pretend it's useful.

***Pannoniae*** "Jesus christ you have so much disdain for such a large set of people." - Correct. I kind of hate myself too. If everyone is afraid to say these things or afraid to have a disdain for bad things, nothing will improve. Things improve because some people are pissed off enough to do something about it, not by just going along with whatever.

"Have you ever considered that you might have a chip on your shoulder about Haskell adoption?" - No. I barely use Haskell myself. This could have been written about any other language, my opinion would have been similar. I don't have a strong emotional attachment to Haskell specifically.

***jamincan*** > Sometimes I wonder if some language communities really are cults. It's the same with Lisp - replace "type system" with "homoiconic" and you have the exact same mantras about ultimate supremacy and "intellectual" superiority. But the proof of the pudding is in the eating and across probably 10 computing devices in my home, not a single one has any Haskell or Lisp programs running on it. Lots of Java, C, C++, Python, Rust, Bash, JavaScript, even Perl (probably) but not a single binary compiled from Haskell or Lisp interpreter to be found. To any reasonable person, that should be a strong signal about the so called "power" and value of their sacred cow.

Xmonad and pandoc are probably the most heavily used Haskell programs. Not necessarily contradicting your overall point, just exhibiting the two examples that someone might reasonably be expected to run into.

***doktrin*** > xmonad

I got rid of xmonad in favor of i3 ages ago - precisely because it doesn't force me to edit a Haskell script just to change a keybinding. Obviously just my personal anecdote, but this 'feature' alone probably accounts for a significant % of users either jumping ship, or never boarding it in the first place.

***mathisfun123*** fair enough - i don't use xmonad but i do use pandoc on occasion. in full transparency: it occurred to me that i am familiar with postgrest, though i've never run it personally. still no lisp programs though!***littlestymaar*** Isn't HN written in lisp?***Pannoniae*** Arc, but I think it counts because it's a lisp-like....***mathisfun123*** and <looks around> what can we conclude from this?***Pannoniae*** that lisp is a perfectly usable language for writing cool and usable things :)***mathisfun123*** what exactly is hn usable for other than aggregating links and text comments?***Pannoniae*** If it wasn't an usable and/or interesting thing, you wouldn't use it...***vaibhavsagar*** I personally use Vim, but Emacs alone is IMO the strongest counterexample to your claim about Lisp interpreters (followed by all the programs scriptable with Guile).***mathisfun123*** I mean I'm aware of emacs but I am also a vim user so I proudly stand by my claim.***yakshaving\_jgt*** Haskell isn't just used for small programs for computer nerds.

Haskell is successfully used in industry, in some cases with billions of users.

https://engineering.fb.com/2015/06/26/security/fighting-spam...

***consilient*** > Basically, the author's criticism is that the language is too powerful, too expressive, people try very abstract things

I think this is a bit unfair. The author's criticism is that people try very abstract things and don't stick the landing. And to an extent I agree, but the problem isn't that Haskell is too powerful, it's that it's just barely powerful enough for too many things. Contra the author, GADTs are not one of those things, and snoyman can pry them from my cold dead hands. But singletons clearly are. So are impredicativity, open type families, type level arithmetic, and Template Haskell. Idiomatic Haskell practically writes itself, but type-level Haskell is at least as hard as C, and it's almost entirely because of how terrible the ergonomics are.

> Third, about backwards compatibility

Strongly agree here, if anything GHC doesn't break backwards compatibility enough. `Num` is an abomination.

***jerf*** I think one of the problems that people who get overexcited about functional languages and all the strong typing and such have is that they generally start out in a bad place. Imperative programs with every flaw in the book, threading by locks, for loops that try to modify the index mid-loop, mutation running rampant, all the bad things they complain about. I've worked in those code bases in industrial contexts, they can be nightmares for sure.

So they discover functional programming, and they just get hammered with so very many different ways of doing things. At first it all seems impossible, but then it slowly unlocks itself, and look! All the problems with imperative programs went away! You didn't even properly perceive them before, but they're gone.

You can't hear me, so my tone here is completely serious. I strongly recommend any professional programmer spend some time with a language like this and attain enough fluency to write real, non-trivial code in it, not just map a few lists and maybe use a monad or two. Get something that hits the network or something.

And the problem is, too many of them stop here. Conventional programming sucks. Functional programming rules. Anyone who doesn't use functional programming languages is a loser still wandering blind in the wilderness.

However, there are in fact a lot of practical problems with functional programming languages too, many in the original blog post, others expressed elsewhere. It can be as simple as, that critical library I need is not available in my obscure niche functional programming language, but alas, in the real world, that's enough to be a determining factor.

Where the religious-level advocates of FP lose contact with reality is that there's a third option: You take what you learned in the functional programming world and come back into conventional languages. And I don't mean "use a few maps and jam a monad in to your language even though it completely fails to fit"; that's actually still completely missing the point of functional programming. I mean, you start writing "conventional" code except you pay attention to mutation. You may not write completely pure code everywhere, but the more you mix in the more characteristics of pure code, the more of the benefits you get. (I do think there's an interesting discontinuity at 100% pure, when you can 100% count on it and then build further on that ability to be sure all code is pure, but you do still get a gradient of benefits the more purity you put into your conventional code.) You pay attention to side effects and start isolating them into units instead of mixing them in. You learn how to multithread with messages instead of memory sharing and locks. You don't drag in inappropriate APIs from a foreign paradigm; you take the fact your eyes were opened, and you write code with those now-opened eyes.

And it is not perfect. You will still occasionally have the original imperative problems. But you will have radically, radically fewer of them, so few that the cost/benefit analysis of using the super-strong stuff becomes very difficult to justify, especially over the advantages of being able to use that library you really need. (And see the library has a mutation problem and wrap it in a way that solves it for you, instead of letting it drag the rest of your code base down, etc.)

I have ridden the mighty moonworm... errr... I have fiddled with the Haskell type system and done some interesting things with it. But by and large they really weren't worth it, not in the sense that they don't solve some problem, but in the sense that back in the conventional programming world, I really don't have those problems anymore. No credit for solving problems I don't have.

This is where I diverge with people complaining about not using functional programming. They are comparing writing imperative crap with writing pristine functional code. In this context I don't deny I'd take the functional code too. But I am comparing writing eyes-open conventional code with writing normal functional code. In this context the advantages are a great deal more muted and it isn't anywhere near the day-or-night level of difference... and I gotta say, pitching me back on full-on functional programming by claiming I just don't get it and I just want to write bad code and enable bad code and in general be lazy and bad is bad advocacy in almost every possible sense of that term.

***Pannoniae*** I apologise for the comparison, as your thoughts are very well-structured, and in my head it almost sounds like an insult. But this kind of thinking ("more discipline minimises the problems related to suboptimal languages") is kind of like Uncle Bob's test-zealotry - i.e. you don't need static typing/AOP/linting/any kind of bug-reducing feature, because you can just write more tests (even if it's painful or spurious)

Sure, it is possible to be very disciplined, avoid messy control flow, be careful with mutation, etc.... then good luck trying to get everyone else to get on board with a very niche coding style, unless if you have the strict authority to enforce it. Sadly, in most cases you don't, so you are stuck with the non-verifiable madness.

In an ideal world, this would work - but in an ideal world, we would also be using better languages with way better guarantees and compile-time checks.

***jerf*** I don't really get your point, other than trying to insult everyone. It's fairly well established that in the real world, strict functional programming languages aren't an option. So who really cares that you keep railing on about them? Until they're an actual option, we are not being "zealots" when we can't use them... not "refuse" to, can't... and, yes, you're just being an insulting jerk with no real options to offer anyone.

If you want to be wistful for a world where we could use better languages... get in line. It's a long one. But stop expressing it in the form of insulting other people.

You are not the smartest in the room, you are not the only one who has grappled with the issues, and if you'd stop going out of your way to insult everyone else, maybe you could listen to and learn from people offering solutions. Even if they aren't perfect; it's hardly as if you're offering a perfect one either. You aren't offering one at all.

***Pannoniae*** Sure, but if we are talking about a place where let's say functional programming is off-limits entirely (note that I wasn't strictly talking about functional programming in my previous posts; I was talking about anything, any paradigm or tool which helps write more sound, more provable, more stable and less buggy programs)... then the "discipline yourself and enjoy the benefits of FP in less capable languages" will not work either, because in that place, they clearly don't recognise the benefits, the language doesn't encourage it either, so they'll just write the average code as usual.

I am not insulting people personally, I am insulting the "worse is better" culture which is prevalent in the profession. No one has an immutable opinion about this, and I do not think that looking down upon an opinion or set of beliefs is a hostile thing. In a (slightly weird) way, it's actually constructive because it might plant a seed of thought in people that it's possible to do better. I'm doing the opposite of looking down at or insulting people here - I am assuming they are rational and capable of thinking, they just haven't encountered a different mindset than the prevalent and trendy one, and that's a perfectly fine thing.

"Developing/using better tools, languages and trying to change developer culture" is a solution. Sure, not a silver bullet or something achievable quickly or easily, but on a smaller scale, it's definitely possible to achieve success.

***yakshaving\_jgt*** > It's fairly well established that in the real world, strict functional programming languages aren't an option.

This just isn't true.

We write essentially 100% Haskell where I work. All day, every day. Have done for years.

We have a "real world" business, with real customers, who pay real money.

***consilient*** > And it is not perfect. You will still occasionally have the original imperative problems. But you will have radically, radically fewer of them, so few that the cost/benefit analysis of using the super-strong stuff becomes very difficult to justify, especially over the advantages of being able to use that library you really need.

This has unfortunately not been my experience. Working alone, sure, I can write acceptably pure code in a conventional language. But generally the issue isn't what I can do, it's what my coworkers will do. Even if almost everyone manages to maintain strict discipline without the support of the language, it really only takes one "productive" cowboy to create a disaster. The biggest advantage of pure functional programming is that taking the path of least resistance produces code that's still sort of ok.

***behnamoh*** This is amazing! Thank you for writing it. Maybe you should write a blog or something.***xedrac*** > lamentations about bad tooling

I think it's less about linters and more about basic tooling for toolchains, dependencies, cross compiling, lsp, etc... compare the Haskell tooling to Rust tooling and it's easy to see how deficient it is.

> Backwards compatibility is simply a design aspect, not some holy commandment you must adhere to all times.

It's also something that has enormous influence on industry adoption.

I adore Haskell the language. I use it daily. But I make no excuses for its many shortcomings. I would love to see Haskell prioritize industry needs over academic purity.

***Pannoniae*** Sure, it might help with industrial adoption, but the more you commit to backward compatibility, the more mediocre your language becomes. (You can't really adapt new things if you have to keep everything around.)

What's the point in creating the N+1th generic blub language?

***xedrac*** What's the point of creating a language that isn't used? If the goal of the language is to academically research new programming techniques, then fine. Other languages will eventually adopt some of the more useful ideas and industry development will improve because of it.***Pannoniae*** Not creating something because people don't want to use it is really myopic thinking in my opinion. That's like not creating art because no one buys it. Deliberately aiming for something worse in hopes of public/industrial acceptance is not a good approach IMO.***runeks*** > Deliberately aiming for something worse in hopes of public/industrial acceptance is not a good approach IMO.

It's not "worse", it's a tradeoff between backwards compatibility and fixups. Backwards compatibility is a really useful feature in and of itself, and the question is whether this outweighs the usefulness of all the tiny fixups. In my opinion it mostly does.

***Pannoniae*** I agree with you, it's a tradeoff. I definitely don't want to say that backwards compatibility is not important..... but when you have N other languages priding themselves on that, what is the value add in creating another stagnant but stable language like that?

I think the reason why this is not emphasised more is because breaking compatibility is a visible cost (program broke, time to investigate and fix it) while the pains from maintaining backwards compatibility is an invisible cost. (you can't easily quantify the wasted hours and extra bugs from unintuitive/broken language features) This is kind of a natural consequence of the incentives, so I understand why it happens. I'd be happier if it wouldn't.

***AnimalMuppet*** Some programs live for decades. Some programs get worked on after the original authors are gone. Code is written once, but (if the program is worth keeping) read many times. Optimizing for "easy to write" is optimizing the 10% and ignoring the 90%. Optimizing for "easy to read and understand by someone who is not the original author" is critical for important, long-lived programs.

It's not "laziness". It's understanding that maintaining software long-term is a really significant problem.

***HelloNurse*** Even when the maintenance is short term, there is also optimization for making the expected type of changes quick and/or easy to perform, something that appears particularly hit or miss with language specific problems: having to fit monad square pegs into typeclass round holes when two libraries need to be used together, unexpectedly needing a significant increase of sophistication and/or boilerplate to support a generalization that should be small, finding a compromise between conflicting preludes or incompatible extensions.***erik\_seaberg*** The thing to optimize for is letting experts communicate with each other clearly and concisely. I’m only going to be a newcomer to a system for a short time, and any work while I'm ramping up is not going to be very valuable.***ceddiexd*** Why use a programming language at all? Why not write everything in machine code for that matter?

The point is that Haskell's position on the abstraction spectrum is no less arbitrary than that of Go or C++. You cannot divorce the relative advantages and disadvantages of various levels of abstraction from the realities of maintenance, development time, readability, complexity, and yes, the fact that not everyone is as big brained as you, because by not writing machine code, you have already acknowledged that these realities matter.

I can just as easily come up with my own programming 'language' where there's a 1-to-1 mapping between a countably infinite set of characters and every unique Turing machine. Want to write a program? Find the right character. This is the most terse, most elegant programming language ever, but clearly helps no-one.

The best abstraction is the one that is the most universally understood.

***bigbillheck*** > Sure, your code will compile in 5 years most likely without changes,

Five years goes by a lot faster than you might think.

***dncornholio*** What are we even doing here.***rvz*** [flagged]***js8*** I have only skimmed the article, but IMHO this is a pretty bad summary. (For those who want a summary, the reasons are also outlined in the article.)

## Coffee in a Can

SEP 10, 2023 6 one-from-nippon.ghost.io

Coca Cola's Georgia Canned Coffee Source: sinkdd

*Life’s inconveniences, when they happen to you and me, are just that. Inconveniences.*

*When they happen to some people though they end up becoming multi-billion dollar businesses.*

*This is the story of Japan’s canned coffee.*

One finds at least one vending machine in almost every street in Japan. In every vending machine one row is dedicated to just one product: canned coffee.

*[The bottom row on this vending machine is all coffee. Blue buttons are cold coffee and red ones are hot Source: Kevin Dooley]*

Canned coffee, as it says on the tin, is ready-to-drink coffee in a can.

There are hundreds of variations of canned coffee and some can’t even legally be called “coffee” in Japan.

Inconveniences and thrifty entrepreneurs

*[Bottled milk coffee Source: こやまうんてん]*

The year was 1969. Tadao Ueshima was at the train station.

In the sweltering heat of the Japanese summer he got himself a bottle of cold “coffee milk” (コーヒー牛乳) from the station’s convenience store, figuring he had a few minutes till the train left.

In those days you would buy your drink, drink it on the spot, and return the bottle. As luck would have it, he mistimed his coffee and his train started to pull out of the station. He hurriedly returned his half-finished bottle and ran to his train.

Now, Tadao had somewhat of a reputation as a frugal man. He hated seeing things go to waste. And he couldn’t shake off the frustration of the coffee. “If only there was a way I could buy and carry my coffee with me…” he reasoned.

Fate couldn’t have chosen a better person for this encounter.

Tadao Ueshima was the CEO of UCC Coffee, which sold coffee and tea in bulk to restaurants. He assembled a team at UCC and gave them an impossible mission: to create a coffee he could buy and walk away with.

Such a thing was unheard of.

The team quickly landed on using steel cans for the coffee. This way people could buy coffee anywhere there was a vending machine. This was probably the easy part.

After all, how hard could it be to make canned coffee? You make coffee, pour it into a can, and seal it, right?

That is what the team started with, but they ran into one problem after the other: the coffee and milk would separate when they left the can sit; the coffee reacted with the steel can and turned into an unappetizing black liquid; the coffee tasted weird when they sterilized the sealed cans.

One by one, they experimented their way out of these problems and finally in April 1969 their first mass-produced canned coffee was ready.

*[UCC Canned Milk Coffee Source: Like\_the\_Grand\_Canyon]*

Remember I said this thing was unheard of? Well, their buyers hadn’t heard of it either and had zero faith that people would want to drink coffee out of a steel can instead of choosing a nice, cold bottle of coffee.

Some people even called it blasphemy to stuff coffee in a can. (Yes, I know many of you think adding milk and sugar to coffee is blasphemy already.)

UCC’s employees went store by store, opening the cold canned coffee and inviting owners to try it for themselves. But it did not move the needle much.

They then changed strategy and focused on the 1970 Osaka World Fair, where companies from across the world came to showcase the latest and greatest from their countries. This was a massive, 6-month long event, which over 64 million people attended.

UCC targeted the restaurants and shops around the 330-hectare venue and this is where they got their first taste of success. In just a few weeks, people were clamoring for the cold milk coffee to quench them in the heat of the Japanese summer.

This newfound recognition from the World Fair catapulted UCC and in just one year, their revenues exceeded 10 billion yen.

**“Real” Canned Coffee**

While UCC was busy taking over the world, there existed people who couldn’t take “coffee milk” seriously. One such people was Toshikage Tanida, the CEO of a company called Pokka Lemon that made synthetic lemon juice.

In 1972, Toshikage stopped at a service area on the highway after a long drive and wanted a cup of coffee to freshen up. There was a long line at the service area's only coffee shop and it took him 30 minutes to get his coffee.

You and I would have grumbled and moved on. But not Toshikage.

The experience annoyed him so much that he set about looking for a way to have real canned coffee. Pure coffee, unadulterated by sugar and milk. Hot coffee.

By 1972 vending machines were the main channel of selling canned coffee, but most vending machines only had one mode: cold. Pokka collaborated with a vending machine manufacturer to build a “hot-and-cold” vending machine which could sell both hot and cold beverages simultaneously.

It took them a year to build this, but in 1973, Pokka could legitimately lay claim to creating the first canned hot actually-coffee coffee.

*[Pokka Aromax sugarless coffee Source: David Pursehouse]*

I am drinking a freshly brewed cup of coffee as I write this, and I can only imagine how much of thought, work, and failed experiments went into canning something as sensitive as coffee.

Especially so, when you realize this coffee is heated constantly, for an indefinite amount of time in a vending machine, and still tastes the way it was supposed to taste! Mad respect for the people who pulled this off.

*[Translation: "Actually, this was canned coffee. Georgia European". Seen at a coffee workshop. Source: Hideya HAMANO]*

**Enter Coca Cola and Tommy Lee Jones**

Most canned coffee was sold through vending machines and thanks to Pokka, Japan now had hot-and-cold vending machines. Guess who had more vending machines than everyone else in Japan? Coca Cola.

In 1975 Coca Cola entered the game with their “Georgia” canned coffee, named after the home state of the Coca Cola Corporation. Coca Cola’s sheer brand might and vending machine network gave it meteoric growth. So meteoric, that to this day, Coca Cola Japan’s best selling product is not Coca Cola. It’s Georgia coffee!

Suntory, another beverage maker jumped into the fray in 1992 with their “BOSS” brand.

In 2006, they hired Tommy Lee Jones to be the face of the brand. Tommy was cast as an alien exploring earth, trying to understand the life of everyday humans.

Suntory wanted to paint BOSS as a coffee for the hardworking people on the street and with their ad campaign they succeeded spectacularly. Watch with the subtitles turned on and I think you will see why.

https://www.youtube.com/watch?list=PL4F3C1016A2216AB3&v=39MILG4txBk

The success of canned coffee rests on two pillars – cans and vending machines. How these coffee cans are made is a rabbit hole of its own if you’re into the sheet metal forming kind of thing. The vending machines too deserve a whole other article of their own. In fact, if you think we should write about Japanese vending machines next, comment down below!

To the people still shaking your head that coffee in a can can’t possibly be good, it can.

I will leave you with this video of a guy who flew to Italy and recorded the reactions of strangers (including a pair of baristas) drinking canned coffee:

<https://youtu.be/LjpMEu_IHKo>

**Coffee in a Can**

330 points by kizunajp 6 months ago | 208 comments

***anotherevan*** In the early to mid nineties I worked in Tokyo outer suburbs on and off for a bit - working ridiculous hours. One fond-in-retrospect memory is in the late evening or early AM hours going up to a vending machine on the train platform, getting a nice hot can of coffee and sticking it and my hands in my coat pockets to keep warm until the train got there. Then once in the heated carriage cracking it open and drinking it to keep me awake for the ride back to the hotel.***255kb*** Having worked in Tokyo during summer, I long for cold coffee cans. In Europe we can only find Starbucks cans full of sugar and milk :(***biztos*** In Bangkok I sometimes grab a Boss Coffee[0] from the 7-11 and I'm inevitably surprised by how not-bad it is.

(I was thrilled to find a serious coffee culture in Thailand that rivals anything outside of Portland or Italy, but coffee at night is not a thing.)

[0]: https://www.suntory.com/brands/boss/

***fennecfoxy*** You can! There's https://en.wikipedia.org/wiki/Mr.\_Brown\_Coffee

German/Euro site: https://mrbrown-coffee.de/. It's made by a company in Taiwan, but I'm pretty sure the local stuff is made in Europe.

I first tried it when I went to Cyprus, they do a "Mr. Black" as well, which is just plain black coffee in a can. It's so freaking good when chilled.

I live in the UK (NZ originally) and so on return I missed it, but found that the greek shops near me readily sell it.

So you should be able to find it in Europe. Not as good as boss coffee etc but still holds a stick, give it a shot!

***paulcapewell*** I've never seen it in the UK, but having seen it in Cyprus, I will have to check out Greek Cypriot stores in London now...***fennecfoxy*** Fio's usually seems to stock it from what I've seen, they have a few locations around North London.***rmbyrro*** Didn't it ruin your sleep?***anotherevan*** With the hours I was working at the time, no.***skhr0680*** They used to make hot drinks really hot just for the purpose you mentioned, but in the last few years they are just lukewarm to save money on electricity***hnben*** I guess also to avoid bodily harm. Iirc the famous hot-coffee lawsuit (about a woman suing mc donalds because the coffee was too hot), was because it was much hotter than it needed to be, which gave her 2nd (3rd?) degree burns (also I think the lid wasn't correctly attached?) and she hat to pay >$10000 in hospital bills.

Thanks to you comment I finally know why the coffee was so hot in the first place.

***geswit2x*** I'd pay to know more about that story lol***anotherevan*** Well one time I broke a tooth. Using my little Japanese/English phrase book I told the concierge at my hotel who took me to a local dentist. The dentist whacked a temporary filling in and via the concierge's rudimentary English told me to get it fixed when I got home.

The only thing he said to me directly, and with no trace of an accent mind you, was, "Open wide, please."

***geswit2x*** hahahahahaahah***version\_five*** In Japanese, tooth is Ha (歯,は)***pard68*** My roommate through all of college was a man from Taiwan. We were and still are good friends. He started dating a girl from Japan around our Sophomore year and the next thing I knew our room was full of cases of canned coffee.

Turns out the girl's father had tried to start up an export business shipping cans of coffee to the US. Didn't work, but my roommate loved it so his girlfriend off-loaded all of it on him, after her father had done the same to her.

***theGnuMe*** This would probably work now... hot canned coffee in airports, specifically, as the Starbucks are always backed up and never open for the very early or very late flights. I would target the major international airports first.

Also my kid is really into Ramune as a soda. Even our almost "rural" edge of the metro supermarket is carrying it.

***version\_five*** I think airports work on a concession model where companies essentially bid to get a spot there, like a mall food court. So the business model is to get a monopoly and then sell overpriced crap, which doesn't lend itself to innovation and makes the market hard to break in to.

I agree it would be awesome to be able to get it at the airport.

***blamazon*** This is changing, slowly - airport leaders today recognize the value of innovation for customer experience and are making moves where possible to back out of such models, plus welcoming local business, although the difficulties of operating in an airport environment does still lend the environment to larger players, or smaller ones operating through a middleman. KATL and KMSP are two that come to mind for me. Entrepeneurs should definitely consider reaching out to airports that are locally managed - often their hands are tied due to contracts, but those will time out someday.***ggm*** Building a vending machine to do hot and cold was a master stroke. There's also a niche for vending machines with a small microwave to heat noodles.***hokumguru*** Perhaps my favorite small channel on YouTube is an indie musician who travels through rural Japan to visit noodle vending machines, accompanied by his own smooth elevator jazz. The vibes are out of this world.

https://youtube.com/@onsenjazz?si=IgB0LTclAFeVY4TN

***jwong\_||| Traveled down a rabbit hole of their videos -- all of them are great! And the music as well. Makes me want to plan a roadtrip.***

***Purchased their CDs to listen to in the evenings until then.***

***|||ricardobayes*** Only Japan can pull it off. In Europe I pretty much never used vending machines as they are so gross and dirty. Even in "clean" countries like Austria or Germany they are like that.***Roark66*** That's a surprise to me. Having lived and travelled across Eastern, Central and Western Europe a lot I never considered vending machines gross. The first vending machine I ever saw must have been a coca cola (or pepsi) vending machine in my primary school 30+ years ago. Then my high school too had a coca cola vending machine and then they were everywhere.

As they became more popular I started seeing them on gas/petrol stations, rail stations and many other places.

However, the kind of machines I talk about always dispense cold cans, sweets, crisps, sometimes packaged yoghurt, but never hot stuff.

Any time I encounter a hot vending machine they are always within some establishment like a gas station where you're supposed to pay first, you get a cup and then you go to the vending machine. The staff of the place is responsible for keeping it clean. However in busy places sometimes there is trash around them (people dropping sugar packets on the floor etc). So perhaps you meant this kind.

***keiferski*** Europe tolerates graffiti and de-beautification of public areas too much for quirky complicated things like these to work, unfortunately.***nmfisher*** Singapore has them too. Never eaten from them so I don’t know what the food quality is like, but they’re never busted or filthy like you’d see in Australia.

You can have a lot of nice things when you have a government (and a society) committed to law and order.

***Renaud*** Clean and sterile.***rustybolt*** What exactly is gross and dirty about them? I have never thought about them that way (but I don't use them because they generally only have overpriced unhealthy crap).***schroeding*** It's very hit-or-miss in my experience. Some vending machines are immaculate, but the exact same type of machine by the same owner e.g. at another station can have their glass broken, have chewing gum put into it's mechanism for weeks, mystery liquids in the output, etc.

Anecdata, but a vending machine at the local rapid transit station (one of those two-sided hot drinks / soups and snacks combo machines) had its output chute filled with (very visible and smellable, probably human) feces for a week after(!) calling the service number and only then the owner cleaned it. Mhhh. Didn't inspire confidence in their general cleaning routine, for me.

I have no idea how our machines stack up against the ones from Japan or other parts of the world in the vandalism department, though, in practise and outside of city centres. They might face similar problems, I don't know.

***rwmj*** Basically never vandalised (and always maintained, filled and working) even in the most isolated and rural places in Japan. I actually wonder about the economics of keeping a vending machine filled in some of the places I've been. How is it possible to have people driving miles to refill them and still make money.***bbarnett*** Maybe 3G notification if the machine is borked?

Alternatively, if a company operates for decades, they can make a very, very stable product. Only bugfixes. I'd imagine some of these vending machines are manufactured like that.

Which means, a weekly drive by is enough to inspect, refill, drive on. I used to deliver meats to very rural locations, 10 deliveries over a 300+ km circular route from/back to the packing plant. At 80km/hr, the stops are the big slowdown.

Another way to think of it, is that remote often means little traffic, fewer stop signs, and higher speed limits.

I live 8km from a large city. It's often far faster for me to drive 15km to a corner store in a small nearby town, because I can do 120km/hr the whole way there almost, on my rural road, with no stop signs or lights.

Drive to the city, there are 5 stop lights before the first corner store, that's each way, so chances are high to hit a light. And even one light is 2 or 3 minutes delay.

So rural servicing and deliveries can be quite fast and efficient.

***jstarfish*** > How is it possible to have people driving miles to refill them and still make money.

I wondered that myself, after seeing some unlucky porter hauling cases of tea up to the vending machines at the top of some mountain temple (maybe Nikko?) on foot.

It was like something out of Death Stranding. It defied physics, looked absurd and would have killed him if he slipped on moss.

***fennecfoxy*** Naaaah. Recently I used a great vending machine in Brussels station when heading towards Amsterdam by train, rather than dropping my drink straight down using the curly metal things, it has a little platform that the drink slid into and then it brought it to the slot, avoids it getting shook up.

I think the core issue is that vending machines outside of Japan lack variety...there's not much in them.

That said in Europe (not really the UK though) you can always find little vending machine "nooks" filled with machines containing alcohol, drinks, medicines, condoms etc. I think the negative perception is because of these - they seem to get left dirty and homeless/general public just mess up the spot.

But when you think about it, that's less vending machines and more Japan's culture of cleanliness and lack of public disorder. Outside of Japan, well, people don't really give a crap unless it's their own property...

***eloisant*** Vending machine are pretty common in France and not gross at all.

It's nowhere near as common as in Japan but you will often find them in train stations or highway rest areas.

***ghaff*** The US has had vending machines for cold soda and snacks forever. Though, without looking up numbers, there's probably been a general trend towards convenience stores and fast food over time. Certainly the vending machines in Japan are more ubiquitous than I think I've seen anywhere else.***wingerlang*** Thailand (well, Bangkok) has had quite a lot of vending machines pop up over the last couple of years. Some in malls, but especially within condos.***blowski*** Yes, when I think of how dirty the microwaves are in shared houses, the idea of a public microwave doesn’t appeal.***astrange*** Is this why Sweden has those giant microwave banks?

https://x.com/yannseznec/status/1492105643649601552

(I don't even want to tell you about the tvattstuga. It's better you don't know.)

***Too*** Nothing to do with cleanliness, on the contrary they can be quite gross also. It’s because eating lunch out is relatively expensive, especially for students. Close to 13€ now after last years inflation craze, whereas cooking at home is more like 2€. It’s not uncommon to see even well paid white collar workers bring food boxes regularly.***ghaff*** Bringing lunches to work used to be a much more common meme in the US. (But maybe it still is among blue collar workers.) Does seem to have faded out a lot though.***bafe*** I guess is the equivalent of Waschküche in Switzerland? A most hated institution which fortunately is slowly dying out***nikau*** Even better it just uses the waste heat from the cooling heat pump so it doesn't cost any extra to run***ballenf*** Total tangent, but why don't homes use these ideas broadly? We have refrigerators, A/C and hot water heaters all in close proximity. Not to mention cool water. Would it really be so hard to get some of the waste heat or cool from those processes into the others?***ghaff*** >We have refrigerators, A/C and hot water heaters all in close proximity.

Maybe yes maybe no. My water heater is in the basement and I don't have AC. It would all have to be very house-specific and customized. And it would tie together what would otherwise be independent systems. A lot of complexity which is usually a bad thing especially at small scale.

***hoseja*** Fridges use minuscule amounts of energy. The added complexity of integration would overwhelm any savings.***arrow7000*** Oh that's genius***ljm*** One specific childhood memory in the UK is going to a public swimming pool and getting hot chicken soup from a vending machine as if it was a drink just like anything else.

I miss swimming pool chicken soup.

***bbarnett*** Fun fact! The entire experience was a trial in sustainability, the water for the soup came, filtered, from the pool, and of course the pool contained a lot of soup, filtered, by human kidneys.

A closed loop of beauty and perfection.

***twobitshifter*** Are people aware of automats? They seem very cool nowadays but fell out of favor. Hot food from vending machines was a thing before it wasn’t a thing. https://allthatsinteresting.com/automat***ehnto*** There's a channel on YouTube that explores some of these, they are so cleverly simple in some respects. Conveyers, servos, more mechanical than digital. But while some machines do all the work, there are also machines that get stocked daily with freshly cooked food that gets kept warm. Just in general, keeping all the machines stocked is likely a massive effort.

If I had to take a jab at the core difference between Japan and other countries, and why this stuff works there; Logistics prowess and manpower. Organised and affordable employees may be another way to word that.

***johnwalkr*** The microwave vending machines are pretty uncommon. Slightly more common, found outside places like school/company cafeterias, is cup noodle vending machines. They simply dispense a cup of dry noodles and then have a hot water tap that you operate manually.***nine\_k*** We should check if these already exist in Japan and / or Korea. They are big on street vending.***numpad0*** That always existed at small scales but there is now an SV startup is doing that(in Japan too!)[2]

1: https://www.youtube.com/watch?v=Y3rGPXXk8VE&t=105s

2: https://www.yokaiexpress.com/location

***ta988*** They exist for pizzas in Europe.***Nursie*** I had a “pizza” out of one of these things in the UK. Very, very bland. Sorta like a really cheap pitta bread with cheese on top.

I was pretty drunk so it didn’t matter, and it was fun to watch it mix flour and water to a dough, top it with sauce, cheese and pepperoni, then cook it in a matter of seconds.

But in the end there’s a reason we don’t usually go from flour-> eating in under 5 minutes!

***eloisant*** If you want easy pizza, don't care too much about the taste and don't want to pay for delivery (or it's not available in your area), just buy a few frozen pizza. You can keep them for months in your freezer.

It's not great but still way better than vending machine pizza.

***bbarnett*** I tried a whole bunch from the grocery store, and focused on finding one with the best crust. I then:

\* only buy 3 or 4 meat pizzas

\* add some cayene and mustard seasoning

\* add my own fresh veggies

This fixes the bland, and meat seems to freeze well, veggies not so much.

And it takes 5 or 10 for the oven to warm, so cutting up some onion, maybe a green pepper passes the time.

Anyhow, I find this method very palatable.

--

As a side note, any kind of meat, especily pork, is really improved with mustard powder when cooking.

***Nursie*** My freezer and oven weren’t on the way back between the pub and my house :)

We always keep a couple in the freezer for especially lazy days, but this was a one-off drunken “ooh, shiny? Also melted cheese...” moment.

***ido*** If the problem was that it’s bland wouldn’t simply seasoning better solve it?***dzhiurgis*** We have ones selling live worms (fishing bait) in Lithuania***vjk800*** Why the hell isn't this used for more stuff. Like you could have a vending machine for hot pizza or lasagne or something.***ghaff*** Food safety would probably be a nightmare and sounds totally disgusting. Automats did used to be a thing in some US cities but they pretty much gave way to fast food restaurants/convenience stores as we know them today.***williamdclt*** Pizza vending machine are a thing, there’s a few here and there in France***marklyon*** I keep Kirkland canned Cold Brew stocked in my office. We have great hot options at work, but not iced ones. But we do have ice. It's bitter, it's unsweetened, but it packs a good amount of caffeine and is incredibly convenient. Even took a case with me on a cruise.

I have enjoyed UCC in the past; It's good, but I prefer my coffee without sugar or cream. I didn't try any of the hot canned options while in Japan, but did absolutely love some of the choices for other products I found in various vending machines.

***creamyhorror*** UCC has had black unsweetened canned coffee for quite a while (UCC Black 無糖 sugarless).

Sugarless Japanese canned coffee tends to taste rough because it often includes some robusta beans, which are cheaper and more bitter-tasting. Which is why I prefer the less-sweet (微糖) variants, or with milk/creamer to balance the bitterness. The exceptions are the special cans advertising coffee from a specific country picked by some champion barista or other, which are better, but of course nowhere near as good as a freshly-brewed cup.

What's different about mass-produced coffee in Japan is that it leans heavily towards only mildly bitter coffee, not the hard bitter stuff you might get elsewhere. Convenience-store machine-brewed coffee is almost always pleasant and easy to drink unsweetened, which is not the case in many other countries. Heck, a few years ago the 7-11s here in Singapore brought in 7-11 Japan's coffee machines, but the beans they use here are far more bitter than the ones in Japan.

***closedl00p*** I agree about the rough taste in general. I've made it a habit to try as many different canned coffees as I can when visiting Japan (and every "black" coffee I saw), and looking back at my notes, I felt like "Premium BOSS Black" was quite good, and "UCC Black New Grounds Fruity Blend" and "UCC Black Wholly Brewed" were both decent, compared to most of the other black coffees.***anotheruser13*** I used to buy liters of Drink It Black (unsweetened black coffee) or Tasty Club (sweetened black coffee) which I chugged on the 203 bus that I took to Japanese school. Good times!***Tade0*** > t's bitter, it's unsweetened,

To me it seems like an important part of the experience. If it doesn't taste like death, it's not the real deal.

***supportengineer*** I’ve consumed many cans of UCC and it always tastes like someone tapped a cigarette into it. Is it just me?***hayst4ck*** I definitely think that something is very whack about about the Japanese canned coffees. I assumed it was some chemical process to get more use out of already used grounds or some flavor additive.

Finding good coffee in Japan, particularly cold brew/dutch coffee, was hard for me. Finding it at a cafe that doesn't also allow smoking was harder.

I'm still not sure if it was my lack of skill in finding coffee/cafes that I liked or if the coffee/coffee culture was different and not as good.

I found Taiwan and Korea both had high quality and familiar coffee, at least to the US perspective. They also had very nice cafes. Vietnam does great coffee too, but Vietnam is more innovative than refined with their drinks and has a very distinct and unique coffee culture. In Vietnam cold brew was harder to find, but I didn't miss it because the other options were novel and great.

***tkgally*** > Finding it at a cafe that doesn't also allow smoking was harder.

As a long-time Japan resident, I had the same problem for many years. But smoking indoors in public places was finally banned nationwide in Japan effective April 2020 [1].

I don’t know how effective the new rule is in bars and the like, but all of the cafes I’ve been in in recent years have been either completely nonsmoking or have had a separate, walled-off smoking room.

[1] https://www.gov-online.go.jp/useful/article/201907/2.html

***eloisant*** I imagine it was a few years ago, or in country side because nowadays you can find a lot of good independent coffee shops in Tokyo or big cities.***smackeyacky*** I have never had UCC but that sounds a lot like what cheap instant coffee tastes like to me. Initially I thought it might be an auto reaction to old break rooms when people were allowed to smoke so I associated instant coffee with cigarette smoke, but I had a cold brew in a bottle the other day and that same sensation happened.***raincole*** In my experience UCC's cheap instant coffee is better than their canned ones. To my taste of course.***vladgur*** Kirkland canned cold brew is extremely bitter and tastes overly dark and borderline burnt to me. On other hand Trader Joe’s canned cold brew is perfectly medium for my taste***hayst4ck*** I think Kirkland cold brew's problem is the Colombian bean it uses. I remember trying a Colombian pour over and feeling a greater understanding about my dislike of Kirkland cold brew.***RegularOpossum*** Agreed, that Columbian Supremo is roasted beyond what I would even call dark. The Kirkland signature blends are roasted by Starbucks so we know why those are burnt but the Columbian seems to be by someone else.***CobaltFire*** I’m not sure when you were last in Japan, but I know the BOSS machines at least have a couple of excellent black varieties now.***etrautmann*** I’m not sure I’d go as far as excellent but it can hit the spot under the right circumstances.***nine\_k*** If all you need is caffeine, there are pills, and there are candies. Fast and efficient.

Brewed coffee has other upsides, like the flavor :)

***ac29*** Caffeine tablets are way underrated. The equivalent of ~200 cups of coffee in tablet form is less than $10.

(and yes, I love real coffee too but it is far more expensive even made at home from the cheapest grocery store beans available)

***wodenokoto*** It sounds like you actually don’t like them. And all the offices I’ve been at, getting hot black coffee took about as much work as getting ice into a cup.

So why?

***hotnfresh*** Cold brew coffee is the easiest coffee to have on hand in the morning, and, as a bonus, so-so beans make pretty good cold brew.

Big mason jar, one of those coke brew tea/coffee inserts for it that you can find on Amazon, start it about 8 hours before you want it (the night before) and you’re golden. Add coffee, add water, stick in fridge overnight. It’s not more work than the simplest methods of making hot coffee in the morning, short of keurig or instant coffee.

Don’t settle for kinda-bad premade cold brew when you could have kinda-OK homemade for less money and barely more work.

(Incidentally, I drink my hot coffee black, but like about a teaspoon of heavy cream per 16-20oz of cold brew—I find all but the absolute best cold brew is a lot better with a little splash of cream)

***sogen*** Do you have a pic or a link on how to prepare it, and how your setup looks like?

Thanks!

***mixu*** Here's my setup:

- I use two large 64oz mason jars (best price via Walmart):

https://www.walmart.com/ip/Ball-Wide-Mouth-64oz-Half-Gallon-...

- In the past I used a metal filter from IKEA to filter the coffee by pouring into an empty mason jar:

https://www.ikea.com/us/en/p/oeverst-metal-coffee-filter-sta...

In the past year I switched to a mason jar pouring spout that has a built in filter that's at the right granularity to filter cold brew as you pour it into a glass:

https://www.amazon.com/Tea-Spot-stainless-Comfortable-Pitche...

That saves a bit of effort since the cold brew gets filtered when I pour it.

As far as how I prepare it:

I add a bit shy of 1/2 a cup of ground coffee per 64oz mason jar, mix vigorously and let it sit overnight (less than that and the flavors aren't quite yet there).

I don't proactively filter out the coffee grinds since the built in filter takes care of that. When one jar starts to run low, I start another one.

***hotnfresh*** I use filters like these:

https://www.amazon.com/dp/B0CB628TNX/

Get cheap mason jars locally wherever, (wal mart is usually cheap, some grocery stores may have cheap ones), grounds in filter, filter in jar, add water, jar goes in fridge. You can also use them to make cold (or hot) brewed iced tea from loose leaves. There are different sizes of those filters for different sizes of jars. I like the 64oz jars.

Pull the filter out when it’s done to avoid overbrewing/oversteeping. Keeps a few days in the fridge.

Recipes (grounds:water ratio, brew time) abound online. Pick one and try it. I like 10-12 hours with something like 80g of grounds, with this method, for a 64oz jar.

I should get a pour spout lid for the jar, that’d never occurred to me and they look nice.

One downside to filtering at the lid instead of in a basket-type filter is that you have to pour it in another container to stop the brew. Whether that matters depends on consumption patterns and personal preference.

***sogen*** > Pull the filter out when it’s done to avoid overbrewing/oversteeping

I’d never have thought about that! Thanks, appreciate it

***sogen*** Ended up doing a DIY homemade thing with a filter I had, will check the recommendations.

Many thanks! the coffee-water ratio was going to be my next question.

***freetime2*** Not the parent, but have been using on of these Hario cold brew coffee makers for years. Fill up the basket with ground coffee, add water, and let it sit in the fridge overnight. I agree with the parent, there is no need to use expensive beans for cold brew (although I do always grind my own beans). Black cold brew has a bit of a funny taste to me, but with a splash of milk it’s delicious.

https://www.amazon.com/Hario-Mizudashi-Coffee-1000ml-Brown/d...

***sogen*** These past couple years I’ve been drinking my coffee black, will find out the taste of cold brew tomorrow!

Loved the coffee maker recommendation! looks practical and easy to use, thanks

***sogen*** Update: Cold brew is a game changer, I'm a convert.***jmuguy*** Getting hot coffee and tea out of a vending machine when it’s cold out is one of the best experiences of visiting Japan. Particularly since the machines are everywhere.***spike021*** Went to Mt. Moiwa in Sapporo back in Feb 2019, at night. It was super cold (to me as a Californian). Maybe 20's or 30-ish F.

Nothing like going inside the observatory at the top and grabbing a piping hot bottle of Royal Milk Tea from a vending machine.

***aliasxneo*** Yes! I was first introduced to this during a cold January while having to stand watch in the port (ex-U.S. Navy). The highlight of my morning was that warm can dropping down and warming up my hands for the next ~30 minutes.***quickthrower2*** In Australia they are making barista coffee machines that make reasonable good Lattes! (and definitely better than starbuccks). I think it was Coles “Urban Coffee”. Not as amazing as a light roast single O from a coffee obsessed cafe. But better than a lot of coffee shops!***esfandia*** In Thailand they have these Tao Bin barista coffee machines: https://www.tao-bin.com/en/en

Great UI: there's a countdown to when your order is going to be ready, and while you're waiting the next person can also enter their order.

***supportengineer*** I saw a barista robot in SFO but didn’t have time to try it.***quickthrower2*** Would love to see that. In theory a robot could make perfect coffee, latte art, etc.***kleiba***

manojlds 6 months ago | parent | prev | next [–]

How is it different from, say, a Costa Express machine?***closedl00p*** These are outdoor roadside vending machines, sometimes out in the middle of nowhere (or at least, not near any convenience stores). Driving down a quiet rural road on a cold morning and stopping for a hot coffee in a can is a nice break.***donatj*** I drank sooo much canned coffee during my time in Japan. I really wish it was regularly available in the US. You can get Starbucks drink nonsense but I want real, decent god-fearing black coffee.

You can get the Japanese stuff at Asian markets near me, and I do regularly, but it’s not nearly as convenient as it being in convenience stores.

***ghaff*** You can get black cold coffee in larger containers in supermarkets but, yeah, in general the canned/bottled cold coffee you can buy is oversweetened and overmilked for my taste.***amelius*** Is this iced-coffee, or is it somehow heated?***waste\_monk*** Apparently the self-heating is chemical, triggering whatever mechanism mixes a quantity of quicklime with water, and the result is exothermic.

I tried a self-heating coffee once and to be fair it did heat up pretty well, but the coffee was overly bitter and kind of gross. With the right coffee I could see this being quite nice.

***Gunnerhead*** I remember in Japan they had both! Pretty cool.***brewdad*** These are aluminum cans, I presume? Is the “hot” coffee more like lukewarm coffee or is this something that one would only be consumed in the winter when one is wearing gloves?***kizunajp*** Yes, nowadays all cans are aluminum though they used to be made of steel. The hot drinks are hot hot, not lukewarm hot.***donatj*** I only ever had the cold variety in Japan, but I can imagine the warm variety being nice as well.***throw0101b*** Tommy Lee Jones has been doing commercials for Boss coffee for a while:

> Since 2006, Tommy Lee Jones has appeared in a series of Boss coffee ads. Boss is a Suntory brand of hot coffee that is sold canned (hot or cold). In the commercials, Jones is an alien visiting Japan. He always has a different job—such as train station employee, taxi driver, or a tanned host—as he lives in his new surroundings.

\* https://kotaku.com/this-man-loves-japan-his-name-is-tommy-le...

\* https://www.youtube.com/results?search\_query=tommy+lee+jones...

***kmoser*** More interesting to me is the logo's resemblance to William Faulkner https://unseen-japan.com/boss-coffee-japan-history/***the-dude*** So I really tried : I googled and I watched the whole of 8mins of video.

But what the hell is a tanned host?

***TheDong*** In the 8 minute video they link, you're looking for the commercial at t=179: https://youtu.be/Lv2l1JKuxxY?t=179

"tanned" here is "skin-tanned", as in "tanning salon". "host" there is like "hostess club hostess", but male. You can read what "host" means on the wikipedia page about those: https://en.wikipedia.org/wiki/Host\_and\_hostess\_clubs#Host\_cl...

Host club's hosts are often tanned, it's part of the stereotype. The beginning of the commercial shows him as a "host" (as can be seen by him being a host inside a host club, entertaining a woman), and the end shows him as a "tanned host".

***the-dude*** Thank you. The world is amazing.***m3kw9*** I must have watched 10 of them and there was still more***theGnuMe*** I wonder if this inspired "Men in Black" or having him in the movie.***netsharc*** MiB was released (hold on to your chair) in 1997 and the ads are from 2006 onwards (although the clips I've seen on YouTube seem to be VHS rips).. so, no?***kristianc*** Fond memories of working on the Georgia Coffee account when I was in Japan at the start of my career - insanely profitable drink for Coke as there’s very little distribution cost, simply strike a deal with the bottlers and boom you have 200m units in front of people.

This means that you can have insane variety - 20/30 different types of this canned coffee are not uncommon at all and the Japanese consumers are absolutely hooked on it.

***ca\_tech*** I think another wonderful rabbit hole to look at when talking about coffee in a can is the ongoing quest for the self-heating drink can.

https://theconversation.com/self-heating-drinks-cans-return-...

***postcert*** One aspect I rarely seen mentioned is The price of these canned coffees in Japan. I remember most ranging from 100-250 yen ($1-2 when I last visited). As a tourist that was the biggest selling point for me, getting pretty good coffee warm or cold in seconds.

Sadly every time I see some mention of UCC or Boss coffee I spend a few minutes browsing to see if it can be imported near that price. Lately though it has been getting better with occasional sale on Amazon making them a nice treat.

***dzhiurgis*** > tastes the way it was supposed to taste

It really isn’t that great. Sure it will make do if you are deprived of it in a country where coffee culture is basically non-existent and being warm and accessible everywhere is super nice.

But try it when you are home and it’s pretty vile. I prefer instant one most of times (unless I wish convenient packaging i.e. day in the beach, etc)

***brap*** >You and I would have grumbled and moved on. But not Toshikage.

In many of these sort of entrepreneurship stories, this is the key takeaway. It seems like one really has to disable their auto-grumbling setting and enable something else instead. Of course this is easier said than done.

Is this an innate ability? Something that can be trained?

***user\_7832*** I think it comes more easily to some people than others but I see no reason why it cannot be trained. In my personal case (I find myself often finding and wishing I could solve such problems) I don't have a very high threshold of accepting the "existing state" if it's not comfortable for me.***mattl*** https://youtu.be/3acm7j9k\_1w — Twin Peaks cast did a Georgia coffee commercial***earthboundkid*** I still can’t get the Boss jingle out of my head 20 years since I heard it on TV.***RegularOpossum*** More classic Cooper I was unaware of!***michaelbuckbee*** The article mentions that seeing a collection of historical Japanese vending machines would be neat: there's a great episode of "Strange Parts" on YT that's an engineering focused visit to an informal Japanese vending machine museum.

https://youtu.be/gzxW3B3\_Pak

***okonomiyaki3000*** The Tommy Lee Jones campaign was wildly successful and extremely long-lived but the post makes it sound like it launched the brand. This is not true. Boss was around for some time before that campaign.***kizunajp*** Author here. Thank you for pointing this out! I have updated the post.

Boss was launched in 1992 (which was already a long time after canned coffee had taken off in the 70s) and only hired Tommy Lee Jones in 2006.

***giantrobot*** I discovered and fell in love with Japanese canned coffees when I moved to the Bay Area. There's lots of Asian markets there that carry them.

They are not great coffee nor are they bad. They're mostly just good. Boss and UCC and other brands I can't remember have options with different roasts and brews. In a sufficiently stocked store you can probably find a coffee you like.

Edit: accidentally a word I used to buy a bunch and put them in a paper bag in the shared office fridge. I stuffed the bag in the back so no one ever disturbed it. I still dream of being able to grab one from a vending machine to get the full effect.

***spike021*** Boss Coffee, Rainbow Mountain blend is my favorite. Really hits the spot.

I live in the Bay Area as well and I've found that both Mitsuwa and Nijiya markets haven't had them in stock nearly as much as pre-COVID; guessing some logistics/supply chain situation that still hasn't quite recovered.

I'm going back to Japan next month though, so my cravings will finally be satisfied!

***astrange*** My favorite Japanese brand is Georgia Emerald Mountain coffee, by Coca-Cola Japan, because Georgia doesn't make coffee or have anywhere called Emerald Mountain.

There's also a crepe stand in Akiba called "Atlanta Crazy Crepes" and I always wondered if it was somehow related.

***spike021*** Heh there's also a crepe stand in Harajuku called Santa Monica Crepes, if I'm remembering correctly.

I think I've had that Emerald Mountain one too.

***laundryfolding*** > Tadao had somewhat of a reputation as a frugal man. He hated seeing things go to waste. And he couldn’t shake off the frustration of the coffee. “If only there was a way I could buy and carry my coffee with me…” he reasoned.¶ Fate couldn’t have chosen a better person for this encounter.¶ Tadao Ueshima was the CEO of UCC Coffee

Does this still exist? It seems like most people I know who break into even slightly high earnings and attain some level of affluence are in it primarily for the appearances associated with affluence—signalling that they're above thinking about stuff like this.

***nrp*** I suspect this is a reasonably common origin story. Plenty of people hate waste, and the ones who have some level of wealth or power just happen to be more readily positioned to act on solving it.

For what it’s worth, I was able to start Framework to combat e-waste because I was financially in a position to do so.

***Guvante*** There are plenty of wealthy frugal people, they just don't get talked about much since they aren't as exciting.

Also often people who have tons of wealth start divesting it once they realize they have too much. And again a "could have been a dozens of millionaire but just a millionaire" doesn't make engaging content.

***alasdair\_||| Warren Buffet still lives in the same smallish house he bought in the 60s.***

***The CEO of Ikea famously uses both sides of each sheet of paper on his desk.***

***|||laundryfolding*** > Warren Buffet still lives in the same smallish house he bought in the 60s.

50s, not 60s. And it's a bad counterexample. Warren Buffett is 93 years old. (Or to put it another way, Warren Buffett is old enough to have bought a house in the 50s. I'm discussing the differences between then and now: people who \_might\_ be 50, but definitely weren't alive in the 50s, let alone buying a house.)

***socrates137*** I’d never heard this about Warren Buffet and so looked it up out of curiosity. I’m going to be honest, I hesitate to call it “smallish” for the average American (around 6500 sq ft). But I’ll admit it’s smaller and more modest than most homes owned by billionaires. At least according to my brief search, it’s the only house he owns, too.***theGnuMe*** He's remodeled it many times and put on additions (I think).***dukeyukey*** "Smallish" his Omaha house is about triple the size of the average modern American home, and he bought it in 1958 when homes were much smaller than today!***Ekaros*** I think there is cultural divide between manufacturing, retail and such industries and tech or others. In first category being frugal and efficient makes lot of difference in long run. And in the later signalling has lot of value specially when VC is involved, but also later.***laundryfolding*** There's certainly a cultural divide between industries, but I don't see frugality showing up in the discrepancies. I don't live in the Bay Area, and most people in my sphere of acquaintances who live on the middle-class+ plane aren't getting paid to write software (or trying to) let alone chasing VC money, and none of the people that I interact with most frequently are. (Anecdotally, the software people actually seem more willing to voice positions that align with frugality. Developers are notorious for not wanting to pay for tools, for example.)***radicaldreamer*** Excellent (and mediocre) canned coffee is a Japanese institution***ETH\_start*** I never had these hot bottled/canned beverages, despite spending considerable time in Tokyo's subways. I was always concerned about what chemicals could leach into the beverage from the interior of the container, with the prolonged exposure to hot liquid.

Heat in general is not good for foodstuff, as it generally breaks down nutrients and creates free radicals.

***keiferski*** It’s my impression that the coffee is stored cold and only heated up when you order it…?***ETH\_start*** I've checked and it is not stored cold. It's stored hot so it that it can be immediately dispensed. I'd be happy to be proven wrong.***keiferski*** Interesting, yes I can see why that might be a health issue. But these things have been around for decades now, so presumably they’ve solved this problem.

Edit: I found this:

IH vending machines

This is a type of vending machine equipped with Induction Heating. They store canned beverages at normal room temperature (or lower temperature) on standby and rapidly heat up a selected drink during the process of dispensing. It is possible to raise the heat up to 140 degrees or so in less than one minute in the case of a 250 ml can.

https://www.japanesewiki.com/culture/Canned%20Coffee.html

***ETH\_start*** I have low confidence in society's ability to solve these kinds of problems—where the consequences only emerge years afterwards—in a timely manner.

Interesting find on the IH vending machines. I would buy canned coffee from one of these.

***tdeck*** > In those days you would buy your drink, drink it on the spot, and return the bottle. As luck would have it, he mistimed his coffee and his train started to pull out of the station. He hurriedly returned his half-finished bottle and ran to his train.

> Now, Tadao had somewhat of a reputation as a frugal man. He hated seeing things go to waste. And he couldn’t shake off the frustration of the coffee. “If only there was a way I could buy and carry my coffee with me…” he reasoned.

Something about this doesn't make sense to me. Why not just return the bottle to a different machine, or bring it back the next time he was in the station? It seems to be like a bottle is actually easier to carry with you than a can, since a bottle can usually be resealed (even crown closure bottles you can push the cap back on).

***kizunajp*** Author here. This was actually something that puzzled me too. There must have been many steps between "Want to carry my coffee with me" and "Let's stuff it in a can" that I wasn't clear about. So I did some more digging (but didn't put this in the post because I thought it'd make it very long).

I don't have the definitive answer, but a few things to consider:

- The bottles might have been resealable, but you would have to be worried about leaks for as long as you had the bottle with you. FWIW, the bottles from that time looked like this: https://www.townnews.co.jp/0607/images/a001046055\_01.jpg

- Someone going on a long-distance train might not be able to bring the bottle back to the same station

- The store at a different station might not take the bottle from you since shops usually pay a bottle deposit to their suppliers and own the (fungible) bottle

- In many countries with bottled drinks the store charges you a bottle deposit that's much more expensive than the drink. Perhaps the frugal CEO did not want to pay that.

- Or even if he wanted to, did not have the time for that transaction given the train started pulling away

***tdeck*** Wow thanks for the reply!***theelous3*** I find it interesting that people associate italy with good coffee. Given their reputation there the coffee is poor. Espresso is typically burned quite badly and praised as strong.

Northern europe, Ireland, and the UK have far better coffee. Also Oz. At least in my experience.

Also, to briefly comment on the oz conversation happening here - perth does have fantastic coffee. My favourite cafe ever is a little place called issacs in rocko.

***grokkedit*** you're completely right. but most italians will never admit it, shunning all of the alternatives.

the strange thing is, we are used to different grapes varieties that are used to produce different wines, and everyone knows which tomato goes in a sandwich, which ones are better in a caprese, which for tomato sauce. but we are all fine with not knowing which kind of coffee we drink, and not knowing how it was treated...

that's why in Italy it's really rare to find someone that drinks coffee as it is, usually we add either sugar (or honey) or milk in order to mask the fact that coffee here tastes too burnt and bitter.

only in the past few years specialty coffee is starting to appear in the bigger cities, and I'm finally getting used to drink coffee after a whole life of tea with milk and the occasional macchiato caldo (espresso with a dash of hot milk), unless when abroad

***lemper*** i tried to acquire the taste coffee and honey. never succeeded. though it should be noted that I used acacia honey which is quite acid itself. so yeah, it was a dumb exercise. now I got no time to explore coffee anymore. as long as the coffee tastes bitter and a bit sweet AND goes well with cig, then I'm fine with it.***joshvm*** Australia and New Zealand have probably the best coffee (on average) of anywhere I've been, if you like milk drinks. The UK has a lot of fantastic coffee shops, but it's still overrun with Costas and a lot of places have added flat whites to their menu, even though they can't make them well.***theelous3*** Costa and insomnia should be shuttered as a public gratuity. Absolute muck from each at their very best.***paulcapewell*** Had to look up Insomnia - hadn't heard of it before in the UK. Seems to be only in Ireland and across the English midlands. https://www.insomnia.ie/nearest-coffee-shop***cylinder714*** Sacramento has Chocolate Fish Coffee Roasters, founded by two fellows from New Zealand: https://chocolatefishcoffee.com/***speed\_spread*** Nowadays Italian coffee is more about the machines and various methods of extraction than the beans themselves. I assume at some point the northeastern African connection provided Italy with better quality coffee than what was available in America but globalization and small batch roasters changed that.***vjk800*** It seems that what is considered good coffee varies greatly. Here in Finland the traditional way to make coffee is very lightly roasted, which makes it quite acidic. Many foreigners don't think very highly of this type of coffee, but people who are used to it often prefer it to "stronger" tasting medium or dark roasted coffees.

I don't know it there's any objective way to determine which coffee is "good" coffee.

***myspy*** The bean is everything and brands like Lavazza are chemically tasting coffee beans. Small shops and roasters with good beans produce better products and small shops with good baristas know how to extract a good cup of coffee. I guess those people are everywhere but hard to find.***elyobo*** When I lived in Perth in 2008 the coffee was awful compared to NZ (and to Melbourne, where I ended up afterwards). I lived in Leederville and Subiaco, both with popular cafe strips, but broadly awful coffee.***theelous3*** Perth's come a long way then. I was therw for a few months over the oz summer. Coffee was broadly good and often great. It's spilled over, I suppose!***wdb*** Wondering if they have similar stories about tea. There aren’t many good ice teas in vending machines. Barely taste like tea. Personally I don’t drink coffee but can imagine it’s similar to the tea problem?***johnwalkr*** Drink vending machines in Japan will typically have 3-4 types of unsweetened iced tea (such as black tea, oolong tea, barley tea, green tea and jasmine tea) and sometimes sweet/flavoured black tea and English-inspired milk tea. In winter 1-2 of those will be offered hot, too.***rvba*** I dont have any information about canned tea, but I would assume that it exists, since Japan has a whole tea ceremony ( https://en.wikipedia.org/wiki/Japanese\_tea\_ceremony ).***e12e*** Always struck me as a weird descrepency in William Gibson's "Idoru":

> (...) His computer there, a featureless black cube. A shallower shelf of the juice-carton board supported a pale blue microwave, unopened ramen bowls, and half a dozen tiny steel cans of coffee.

> One of these, freshly microwaved, was hot in Chia's hand. The coffee was strong, sugary, thickly creamed. She sat beside him on the lumpy bed ledge, a padded jacket wadded up behind her for a cushion.

I mean, you can't (well shouldn't) microwave a steel can, right?

***throw0101b*** > I mean, you can't (well shouldn't) microwave a steel can, right?

Sharp points are a problem, but one can design things to work:

\* https://www.youtube.com/watch?v=pg\_lqy1p6B4&t=11m20s

***dr\_hooo*** Microwaving steel isn't really a problem, as long as it is not just a very thin layer (can confirm)***e12e*** A (regular) steel beverage can is pretty thin, and comes with sharp edges on the opener ring/tab? (I suppose the tab could be plastic - but they're not now)***ricardobayes*** My buddy brought me one of these from Japan, halfway across the world in a luggage and still tasted good. I drank it cold, never thought I could warm it up. Was an ok cold brew.***WeylandYutani*** The greatest invention of mankind. Never understood why this didn't catch on here. Don't have to time to grab a coffee from Kiosk when I'm waiting for a train.***suction*** Because the taste is not good for European palates.***dukeyukey*** Canned cold coffee is pretty common in the UK, even small shops will have several brands. Never seen canned hot coffee though.***NeoTar*** Back in the early 2000's (maybe 2004-2005) I remember buying a can of Coffee from WHSmiths which was self-heating - I think through some exothermic chemical reaction.

It was also really bad as I recall, more like an instant Nescafé than something you'd get in a coffee shop. I didn't repeat the experience.

***mraza007*** This reminds me of hotshot coffee from sharktank.

I remember trying that coffee at NYU hackathon and actually meeting the guy who pitched that idea at sharktank

***anotheruser13*** The first time I went to Japan, I was wandering around Osaka with a friend. I spied a vending machine featuring a new canned coffee and started laughing. The coffee was called BM.

Granted, coffee has that effect on some people, but I think Pokka was unaware what BM is a euphemism for in English.

Needless to say, that brand didn't last long.

***perebas*** I wanted to live in Japan after this post LOL***m3kw9*** It tastes industrial, but in a good way. When you have nothing else and you need one now, it’s decent enough***AlchemistCamp*** I like UCC black coffee enough that I ordered it off of Amazon when I moved to California for a while.***supportengineer*** It’s available in any local California Japanese market also - Nijiya, Marukai, Mitsua.***couchand*** La Colombe makes a hell of a canned latte if you're looking for something like this stateside.***night-rider*** > Pokka collaborated with a vending machine manufacturer to build a “hot-and-cold” vending machine which could sell both hot and cold beverages simultaneously

How does the 'hot' thing work? Is it kept hot for long periods of time?

***joecool1029*** I miss the UCC Drink It Black coffee cans. I used to see them at Asian grocery stores (Mitsuwa was where I first tried the cans) and would buy a dozen or so at a time. I guess they discontinued the line.***anotheruser13*** That was good stuff! I'd buy it by the liter at the 7-11 near where I lived in Kyoto, and chug it on the 203 "guru guru bus" on my way to Japanese school.***backendanon*** I was in Sasebo, Japan in the early 1980's and remember buying cold coffee in a can from a vending machine similar to buying a can of Coke in the States.***cmod*** This old site: https://cannedcoffee.org/***Waterluvian*** I dunno if this is just me but I find cold beverages just feel colder and nicer in cans than plastic or glass bottles.***PartiallyTyped*** It’s probably the heat transfer from your mouth and the shape of the holding medium [1]. For me, can > glass > plastic. Apparently there’s a certain kind of art to choosing shapes for cold beverages.

[1] https://www.youtube.com/watch?v=vCbYEyfhTQQ

***eloisant*** Well that makes sense, metal have a higher heat capacity than plastic or glass so they stay cold longer***Waterluvian*** I think that might be moot. A can conducts so well that I imagine it pretty much shares the temperature of the beverage at all times. So how much is the heat capacity of the can vs. the liquid? I’m betting the liquid is like 99% of it.***Ekaros*** I would actually expect that thermal conductivity is so much better that they heat up faster. But at the same time sensation of holding one makes it feel cooler.***snvzz*** The problem is getting real-ish black.

With no trace of sugar or sweeteners, nor milk.

***an\_aparallel*** i tried Boss hot coffee in japan, holy moley is it sweet. Cant say I was a fan. Though Sydneysiders can be coffee snobs - im sure this works for 99% of the population***sgu999*** > The success of canned coffee rests on two pillars – cans and vending machines.

I can think of a few more pillars: cheap energy, laziness, and carelessness for the environment

***dannyr*** not mentioned in the post was UCC solved the separation of milk and coffee issue. any resource on this?***kizunajp*** They solved the separation issue by homogenizing the milk: https://en.m.wikipedia.org/wiki/Homogenization\_(chemistry)