Sym·poly·mathesy by Chris Kry

How to Do a TypeScript Conversion

Addressing a very common question: do-it-as-you-go or follow the dependency graph?

Assumed audience (https://v4.chriskrycho.com/2018/assumed-audiences.html): Software developers working with JavaScript and TypeScript, or thinking about and working with gradual type systems in other languages. In particularly: I am not arguing *for* TypeScript or Python `types` or Ruby's Sorbet etc.; I am talking to people who are already interested in adopting them.

Epistemic status (https://v5.chriskrycho.com/journal/epistemic-status/): I led the conversion of a 150,000-line-of-code app to strictly-typed TypeScript back in 2017–2018, and was the primary "subject matter expert" for LinkedIn's adoption of TypeScript across its millions of lines of library and application JavaScript.

One of the most common questions I get from people interested in converting their JavaScript applications to TypeScript is: *How should I approach this?* There are two approaches people tend to think of:

- A relatively relaxed approach: setting `compilerOptions.strict: false` initially, converting files as you touch them, and gradually increasing the robustness of the types by enabling individual strictness flags until you have them all turned on — or some combination of these.

- A more rigorous approach: setting `compilerOptions.strict: true`, and very carefully converting the codebase in a "leaves-first" order, where no module is converted without first having types for all of its dependencies. Making explicit what "more rigorous" probably already implies: this is my preferred approach.

Most developers (myself included, the first time I did this!) are *very* much tempted to do the "just convert a file when you touch it, in loose mode or with lots of `// @ts-expect-error` and `any` scattered around" thing. It seems like the lowest-friction, fastest, easiest pat

- That pattern *usually* works with other kinds of migrations.
- It feels more tractable, in that you can just do it "as you go".
- It actually works pretty well for sufficiently-small codebases — it's very good for <1,000LOC and pretty good for <10,000LOC.[1]

Accordingly, it is also the approach I see most often recommended to people starting out on converting a TypeScript codebase.

Unfortunately…

It's a trap!

You will encounter two big problems when you take the more relaxed, intuitive, much-recommended approach. On smaller codebases, these problems may not matter all that much, but the bigger your codebase is, the more they will hurt.

First, you will end up having to propagate changes to various files over and over and over again:

- Each time you enable another strictness setting, you will see new type errors in many modules. The biggest of these will be

`strictNullChecks` and `noImplicitAny`, but *all* of the strictness settings will catch things missed without them: that is why the settings exist, after all. These are not usually spurious errors, either.[2] Thus, you will have to do *another* pass "fixing the types" for the module each time you enable a new strictness setting.
- If you convert a module but have not converted the modules it depends on, all the types from those dependency modules will be `any`. When you convert those files, you very often find mistakes in the way you were using their APIs. Just like with strictness settings, this means you often end up having to "fix the types" for other modules each time you make this kind of change.

—/ NOTE /—

I scare-quoted "fix the types" here because it is usually "write the types and *fix the bugs*". As I have written before (https://v5.chriskrycho.com/journal/is-typescript-good/):

> …in many cases the complexity was already present in the code base. The TypeScript conversion did not create that complexity: It exposed it. Real-world JavaScript code is often incredibly complicated — indeed, *clever* — in ways that only become obvious when we try to express in types the contracts the code already invisibly assumes. As a result, conversions from JavaScript require complex types far more than code written in TypeScript from the start. Much of the complexity is (permanently!) implicit in JavaScript, while writing out the contracts in TypeScript makes it explicit. That enables better choices: does this particular API actually warrant some complicated types, or should we just keep it simple? Usually: the latter.

Even so, it can *feel* like we are just fixing TypeScript issues over and over again, and I think it is important to acknowledge that.

This kind of thing can be quite demoralizing at a personal level, as you work to "fix types", because you find yourself hitting the same pieces of code over and over again. It can also be difficult to communicate clearly to less technical team members, e.g. designers and product owners who can be quite reasonably confused about why we need to spend time on doing TypeScript things for this chunk

of the codebase *again* — "Didn't we do that a month ago?" Havin
to explain that "Yes, we did, but not all the way" can be frustratir

Second, and maybe even worse, you *cannot* rely on the things you have already converted actually being safe when taking this approach. They *feel* safer than JS types-wise because they are in TS… but they are not, because they have lots of `// @ts-expect-error` and `any` scattered around. It can end up being quite demoralizing and frustrating to have errors coming out of your "but we already converted this!" modules. It also undermines a lot of the promises we make when justifying the investment to our management or partners: "I thought the point of TypeScript was to fix these kinds of bugs, so why isn't it doing that?" As with having to do multiple passes on the same files, having to answer "Well, we converted this to TypeScript, but not all the way…" is deeply unsatisfying.

Finally, the problems described here scale exponentially in difficulty with the size of the codebase. With 1,000 lines of code, these problems are minor annoyances. With 10,000 lines of code, they are a bit of a hassle. With 100,000 lines of code, they are actively demoralizing. With 1,000,000 lines of code… you might just never finish. The friction never goes away, so it requires constant effort to keep it moving and get it across the finish line.

The more rigorous approach of setting `compilerOptions.strict: true` and walking the dependency graph in order means you never have to revisit the file because of increased strictness or newly-well-typed dependencies. What is more, there is a really big upside to the experience when you do a conversion this way. When all your dependencies are already strictly typed, every time you convert a new module it sits on a solid foundation. TypeScript itself can do a fair bit of the work of adding types for you via its code fixes, since it can use the information from the downstream APIs you call. For the parts you have to figure out on your own, you still have to check all the ways the module's APIs are *used*, but you don't have to check all the things it *uses*: the problem space is cut in half.

The net is a double win: every module you convert actuall delivers on type safety, and every module you touch gets easier because its foundation is safe. The process is like a flywheel: every bit of effort you apply speeds up the rest of the process.

This is not a free lunch. It generally requires more discipline and more explicit buy-in from stakeholders. Instead of "just convert a when you touch it (and usually leave it not-fully-converted)" you need to carve out some dedicated time to do the work by tackling a couple modules each week or something like that.[3] Ultimately, though, it makes for a much better experience for everyone involved.

NOTES

1. I suspect this is one of the big reasons this approach is so commonly recommended! Many people with a relatively high profile in the TypeScript community have more experience with TypeScript conversions and maintenance of *libraries*, which are often in this ~10,000 lines-of-code range, where this *can* work reasonably well. ↩

2. It is true that there are sometimes type errors where the runtime code is safe. This is less and less common over time with TypeScript, though, and in my experience the *vast* majority of the type errors surfaced by enabling new strictness flags indicate real bugs in the code. ↩

3. You can still incrementalize this approach. If you have a larger app/etc. broken into a set of smaller packages, you *can* do the "iteratively work within a small library" approach *within* the packages, while avoiding publishing the types until you get them to full strictness. That ends up having some of the advantages and disadvantages of *both* approaches. ↩

POSTED: This entry was published in Journal on November 3, 2023.

Spotted a typo? Submit a correction!

RESPOND: Thoughts, comments, or questions? Shoot me an email (it's way better than traditional comments), or leave a comment on Hacker News or lobste.rs.

ABOUT: I'm Chris Krycho—a follower of Christ, a husband, and a dad. I'm
software engineer by trade; a theologian by vocation; and a write
runner and cyclist, composer, and erstwhile podcaster by hobby.

SUPPORT: If you especially like what I'm doing here, you can buy me a book, or
click the affiliate links in book reviews!