hacker bits

Issue 13

new bits

Happy 1-year anniversary everyone!

When we launched *Hacker Bits* a year ago, not in our wildest dreams did we expect such a warm response from our readers and contributors. So thank you to all of you who have made *Hacker Bits* possible!

In this issue's most highly-upvoted article, Joel Spolsky, who needs no further introduction, tells us all about developers' side projects and its common pitfalls.

Interested in web fonts? Then don't miss Monica Dinculescu's lazy approach to web fonts. And before you get swept up in the AI craze, don't miss Professor Robin Hanson's explanation on the limitations of AI and why like most fads, it's doomed to bust.

And remember, *Hacker Bits* is here for you. We want it to be relevant to your goals, aspirations and resolutions.

So if you have any technology-related resolutions, let us know in this form.

We're here to help!

— Maureen and Ray

us@hackerbits.com

content bits

Issue 13

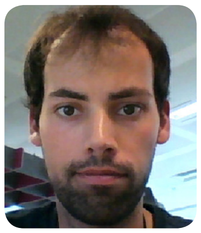
contributor bits

Monica Dinculescu
Monica has been an emojineer at Google for over three years. She works on Polymer, web components, and has probably at least once broken the Internet for you. She will probably eat all of your Oreos, if you have any.

Jerod Santo
Jerod is the Managing Editor of Changelog – a developer-focused media company – and President of Object Lateral – a custom software firm from Omaha, Nebraska.

Joel Spolsky
Joel is a software engineer and writer. He is the author of Joel on Software, CEO of Stack Overflow and co-founder of Fog Creek Software.

Robin Hanson
Robin is econ. prof. at G.M.U., research assoc. at Future of Humanity Inst., Oxford, has Caltech soc. sci. Ph.D., U. of Chicago physics M.S. & philosophy M.A., & 9 years experience A.I. researcher Lockheed, & NASA. His new book is *The Age of Em*.



Ruud van Asseldonk
Ruud is a programmer who likes bare metal systems programming as well as functional programming. He built the native heap profiler in Chromium and he is a contributor of the Rust programming language.

Saron Yitbarek
Saron is founder of CodeNewbie, developer, speaker and community builder passionate about creating the most supportive and inclusive community of coders. She's the host of the weekly CodeNewbie Podcast, new episodes every Monday.

Thorsten Ball
Thorsten is a software developer, writer and speaker. As a result of his early morning deep dives into systems programming topics he published his first book *Writing An Interpreter In Go*.

Chris Krycho
Chris is a Christian, husband, father, software engineer at Olo, runner, and all-around programming language nerd. When he's not blogging, he's often hosting New Rustacean, a podcast about Rust.

Mike Mikowski

Mike is a web application architect, author, speaker, and developer in Silicon Valley. He is currently helping create a great user experience for Perspica's artificial intelligence powered analytics product, used by TechOps and DevOps teams worldwide.

Ted Dziuba

Ted is a founder and technical manager in commerce. He's built teams and products in e-commerce, brick & mortar retail, omni-channel commerce, and 3PL logistics. Check out his blog.

Ray Li

Curator

Ray is a software engineer and data enthusiast who has been blogging at rayli.net for over a decade. He loves to learn, teach and grow. You'll usually find him wrangling data, programming and lifehacking.

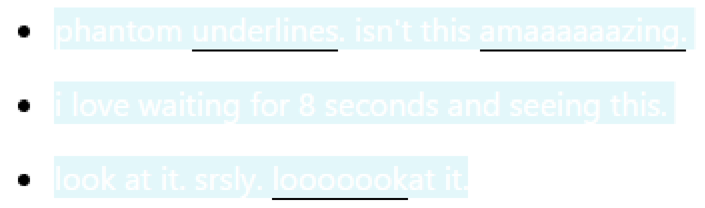
Maureen Ker

Editor

Maureen is an editor specializing in technical and science writing. She is the author of 3 books and 100+ articles, and her work has appeared in the *New York Daily News*, and various adult and children's publications. In her free time, she enjoys collecting useless kitchen gadgets.

Web fonts, boy, I don't know

By MONICA DINCULESCU



- phantom underlines. isn't this amaaaaaazing.
- i love waiting for 8 seconds and seeing this.
- look at it. srsly. loooooookat it.

I spent a week traveling around Taiwan, on my awesome free roaming 2G data plan, and friends, we need to talk about your web fonts. Also cats. They really love cats there. Anyway, the thing about 2G is that I fully understand that it will take me 10 seconds to load a page. What sucks is the fresh rage of the following 4 seconds where instead of content I get phantom underlines, waiting for a slightly-different-sans-serif to download.

Listen: it doesn't have to be this way. You can lazy load your font. It's 4 lines of JavaScript. 7 if you're being ambitious.

Why should you care

I've been brainwashed to *really* care about first paint performance (thanks Chrome Dev Rel 😮), and I've become a big fan of the "do less & be lazy" approach to building things. What this means is that if something is not on your critical path, it probably doesn't need to be the first thing you paint on a page.

Now think about fonts: is the critical path *showing* text, or *styling* it? I'd argue that unless your app is in the 1% it's-all-a-magical-visual-experience bucket (in which case this post is not for you), or we're just talking about the fancy title on your site (which fine, can be slow to paint or whatever), it's probably trying to communicate some content, and ugly content (that you prettify after) is better than no content.

(Real talk: if you don't think rendering text is a critical path, you're whack and we need to have a chat.)

There are two things you can run into when loading a web font:

- **FOIC** ("flash of invisible content") – when your browser sees that you're trying to use a font it doesn't have it paints all the text in invisible ink, waits, and when it finally gets the font, it re-paints and re-layouts the text correctly. [see a gif of this]

I hate this with the fire of a thousand suns, because instead of looking at actual content, I'm looking at bullets and underlines and random text you forgot to style. Neat-o.

- **FOUC** ("flash of unstyled content") – Chrome stops waiting for a web font after 3 seconds (and, recently, after 0 seconds on 2G). What this means is instead of showing you invisible ink, it paints the text in your fallback font. When your web font is finally downloaded, it then re-paints the already displayed text with the new font. [see a gif of this]

Side note: on iPhones, this timeout doesn't exist, so you basically only get a FOIC – you wait the entire time to get from "no text" to "all the text", with no intermediate bail out state.
(Here is the code that I used for these demos, with GPRS and 2G throttling respectively in Chrome. This demo will look super snappy on LTE. Everything is super snappy on LTE.)

Reading material

A lot of people have written about web fonts, and I'm not trying to re-write their posts. Chrome in particularly has been working a lot on improving this, by decreasing the web font download timeout to 0s on 2G, and working on the `font-display` spec.
Here are some links I like:

- the anatomy of a web font and the dance that a browser does to use a web font
- font-display options, and how it affects how fonts load
- font-display: optional and why it's awesome (tl; dr: if you can't do it fast, don't do it at all)
- minimizing font downloads by limiting the range of characters you're loading
- why we should care about web fonts and how to minimize FOIT using JavaScript and a library called Font Face Observer
- voltron solution combining FontFaceObserver, async loading a font bundle and web storage

Lazy loading a font

Personally, I would use `font-display: optional` everywhere, but that doesn't really work anywhere yet. In the meantime, here are 2 super simple ways to lazy load a web font. Again, I don't really mind having a FOUC, since it feels like progressive enhancement to me: display the content as soon as you can, and progressively style it after.

```
<head>
  <style>
    body {
      font-family: 'Arima Madurai', sans-serif;
    }
  </style>
</head>
<body>...</body>
<script>
  // Do this only after we've displayed the initial text.
  document.body.onload = function() {
    var url = 'https://fonts.googleapis.com/css?family=Arima+Madurai:300,400,500';
    loadFont(url);  // hold tight, i tell you below.
  }
</script>
```

There's basically two ways in which you can implement that `loadFont`:

Load the stylesheet (blocking)

This is the simplest way and works great for a simple page. But! Since loading/parsing a stylesheet blocks parsing/painting, this doesn't play nicely if you're loading a bunch of other modules after the document has loaded, since they will be delayed. [demo]

```
var link = document.createElement('link');
link.rel = 'stylesheet';
link.href = url;
document.head.appendChild(link);
```

XHR the stylesheet (asynchronous)

If you care about synchronicity (and tbh you probably should), you can do an async XMLHttpRequest and create a style node with the result. [demo]

```
var xhr = new XMLHttpRequest();
xhr.open(‘GET’, url, true);
xhr.onreadystatechange = function () {
  if (xhr.readyState == 4 && xhr.status == 200) {
    var style = document.createElement(‘style’);
    style.innerHTML = xhr.responseText;
    document.head.appendChild(style);
  }
};
xhr.send();
```

For bonus points, you can take this one step further and rather than creating an inline `<style>`, append a `<link>` like in the previous method, since the browser cache is already primed. If you trust your browser cache. I trust no one.

This is obviously not perfect. It will give you a FOUC on a fast LTE connection, even though if you did nothing, like in the first demo, it wouldn't. The point is that not all of your audience is on an LTE connection, and I want you to think about them when you're working on a site. If you want to minimize this FOUC, Helen Holmes gave an AMAZING talk recently about web typography and performance, where she mentions how you can try to match the x-heights of your fallback font to your target font, so that the FOUC is gentler.

Update: I've built a font-style-matcher that lets you do this matching of the x-heights and widths of the web font and fallback font! Go check it out, it's preeeeetty sweet. ■

TL; DR

Web fonts are okay. They make your blog prettier. They're also slow and kind of an annoying experience, but if you need to use them, use them. Just remember that it's also your responsibility to make your site load super fast, and if you don't, it's totes fair game for people (me) to whine about it on Twitter.

(🎤 to Paul Lewis who had to sit through all my questions and explain basic browser things to me. Again.)

One way to improve your coding

By JEROD SANTO

Author's Note: I originally wrote this for Fuel Your Coding back in May of 2010. Unfortunately, that site is now defunct, so I'm republishing the article here for posterity's sake[1]. I considered updating it for modern times, but I think it holds up well enough as is. The parts that don't hold up are a bit laughable, but oh well. Enjoy!

The most obvious way to improve your coding is to write more code. Everybody knows that. However, another activity which I guarantee will improve your coding is the complete opposite of writing. I will state this as plainly as I possibly can:

If you want to dramatically increase your programming skills you need to be reading other people's code.

Maybe you believe that, maybe you don't. You should. And if you're willing to give it a shot, I believe you will be rewarded greatly for your time.

In this article I will help you choose what to read and give you practical advice on how to go about reading it. If you're already a code reader you may find a few ways to get more from your efforts. If you aren't, you absolutely must read on.

What to read

This is a big decision, and one that is difficult to advise on. I won't simply point you toward code I think you should read, because it really comes down to what you're in to. However, I will provide some guidelines which you can follow to help you choose what to read.

Read code that you rely on

A great place to start is with any plugins or libraries that you are already using.

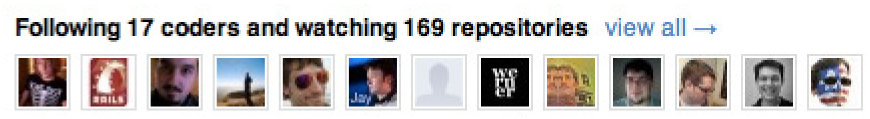
- A WordPress plugin that you really like
- A Ruby gem that you've found useful
- A jQuery plugin that you keep going back to

These are all great candidates for learning. You are already familiar with their public APIs so the barrier to understanding their inner workings is lower. Also, as a user of the code you have an opportunity to add documentation, implement a new feature, or generally give back to the project in some way.

Read code that impresses you

I remember the first time I saw 280 Slides and I thought to myself, "Now that is impressive." I quickly learned that the code driving that site was the open-source Cappuccino project. I tucked that knowledge into the far recesses of my brain and when I eventually came across another impressive app running on Cappuccino I knew I had a project that I could learn a lot from. What have you been impressed by lately? Is it open-source? If so, it's a great choice for reading since the code is likely to impress you as much as the resulting application.

Read code written by somebody you respect



1 You can still view the original article thanks to The Wayback Machine

If you've been coding with open-source software for more than a little while, there are probably other programmers who have earned your respect. I can think of a few developers off the top of my head whose code is downright enviable.
If you don't have a respected developer in mind, it's easy to find one. S/he has probably authored some code in one of the previous two sections (code you rely upon, or that impresses you).

Read code that you can actually grok

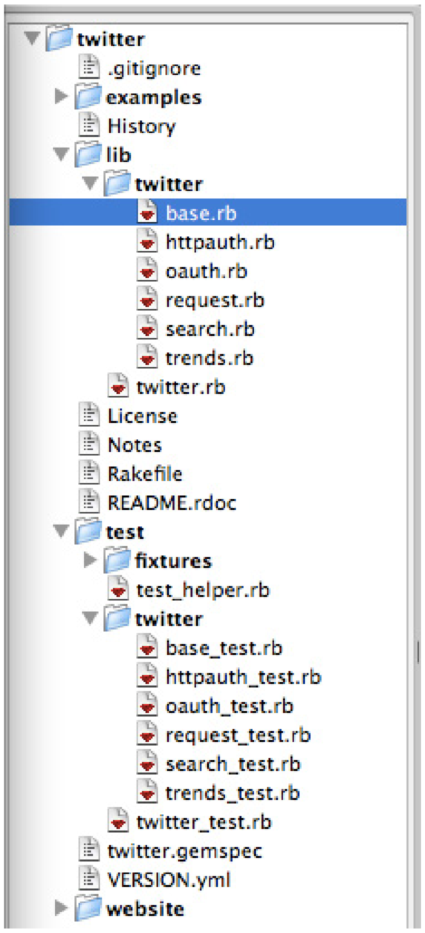
If you're the adventurous type you may be considering diving into a large project like Ruby on Rails, Drupal, or jQuery. I suggest avoiding projects like these for now unless you are an experienced code reader.

Large projects have a lot more moving pieces, and you may end up struggling too much with the concepts to learn anything of immediate value. Confusion leads to discouragement, and larger projects more likely to confuse and discourage you in your reading. The advantage of picking a small project to read is that you can hold the entire program logic in your head at once. This leaves you with just the details to discover and learn from.

How to read

Now that you've chosen some code to read, what is the best way to go about reading it? I've read a lot of code in my days, and I can suggest a few ways to maximize your ROI.

See the big picture



```
▼ twitter
    .gitignore
  ▶ examples
    History
  ▼ lib
    ▼ twitter
        base.rb
        httpauth.rb
        oauth.rb
        request.rb
        search.rb
        trends.rb
      twitter.rb
    License
    Notes
    Rakefile
    README.rdoc
  ▼ test
    ▶ fixtures
      test_helper.rb
    ▼ twitter
        base_test.rb
        httpauth_test.rb
        oauth_test.rb
        request_test.rb
        search_test.rb
        trends_test.rb
      twitter_test.rb
    twitter.gemspec
    VERSION.yml
  ▶ website
```

I'm going to assume that you at least know at a macro level what the code you're reading accomplishes. If not, I suggest reading the project's website, tutorials, documentation, and anything else you can get your hands on except the code.

Okay, with that cleared I suggest your first step is to wrap your head around the project's structure. This is a variable amount of work depending on the size of the codebase you've chosen, but anything larger than one file will require a little bit of time.

First, note the file structure. This step is aided by an editor that has a folder hierarchy view like TextMate. For example, here is a nice overview of the Twitter Ruby gem.

The goal with this step is to just get familiar with the source. Find out which files include/require/load other files, where the bulk of the code is, the namespaces used if any, and things of this nature. Once you've got the big picture you'll be ready to dig into the details.

Document your findings

Reading code should not be a passive activity. I encourage you to add comments as you go, documenting your assumptions and your conclusions as you begin to understand the program flow. When you first get started your comments will probably look something like:

```
# I think this function is called after 'initialize'
# What does this equation even do?
# Pretty sure this variable loses scope after line 17
```

As your understanding progresses you can remove the little breadcrumb comments you left yourself and perhaps write more meaningful and authoritative comments that could possibly be committed back to the project.

Use the tests, Luke

Hopefully the project you've chosen has a test suite. If not, you can skip this section altogether (or find one that does).

Tests are a great place to start whenever you read somebody else's code because they document what the code is supposed to accomplish. Some tests are more informative than others, but no matter how well written, you'll often find the programmer's intent in the tests much more easily than you'll find it in the implementation. While you're reading, try to get the test suite to run successfully. This will make sure your development environment is configured properly and will make you more confident when making changes.

Execute, change stuff, execute

Who said reading code had to be hands off? You'll really start to understand things once you've broken everything and put it back together again. Remember those tests you got passing? Make them fail, add some more, or try changing the implementation without breaking them. Try adding a small feature that you think is cool, or setup project-wide logging so you can print output at various stages of the code. Is this still reading? Absolutely, but at this point it's more of a choose your own adventure than a mystery novel. And that's a good thing!

Rinse and repeat

Once you finish reading one codebase, pick another one and start the process over again. The more code you read, the better you get at reading it and the more you get out of it in less time. I think you'll find that the ROI increases quite quickly and that it's actually a very enjoyable way to learn.

Where to start

The single most influential factor in my code reading is GitHub. The site makes it so easy to find new projects and great coders that you're doing yourself a disservice if you're not leveraging it. I suggest starting on GitHub and reading code right on the site until you find a project you know you can learn from. Then git clone that baby and get to reading! ■

How about you? Do you read code as a learning tool? Which projects would you recommend to others? Read any good code lately?

Developers' side projects

By JOEL SPOLSKY

Pretty much 100% of developers working for other people end up signing some kind of "proprietary invention agreement," but almost all of them misunderstand what's going on with that agreement. Most developers think that the work they do at work belongs to their employer, but anything they work on at home or on their own time is theirs. This is wrong enough to be dangerous.

So let's consider this question: if you're a developer working for a software company, does that company own what you do in your spare time?

Before I start: be careful before taking legal advice from the Internet. I see enough wrong information that you could get in trouble. Non-US readers should also be aware that the law and legal practice could be completely different in their country.

There are three pieces of information you would need to know to answer this question:

1. What state (or country) are you employed in?

There are state laws that vary from state to state which may even override specific contracts.

2. What does *your* contract with your employer say?

In the US, in general, courts are very lenient about letting people sign any kind of contract they want, but sometimes, state laws will specifically say "even if you sign such and such a contract, the law overrides."

3. Are you a contractor or an employee?

In the US there are two different ways you might be hired, and the law is different in each case.

But before I can even begin to explain these issues, we gotta break it down.

Imagine that you start a software company. You need a developer. So you hire Sarah from across the street and make a deal whereby you will pay her $20 per hour and she will write lines of code for your software product. She writes the code, you pay her the $20/hour, and all is well. Right?

Well… maybe. In the United States, if you hired Sarah as a contractor, she still owns the copyright on that work. That is kind of weird, because you might say, "Well, I paid her for it." It sounds weird, but it is the default way copyright works. In fact, if you hire a photographer to take pictures for your wedding, you own the *copies* of the pictures that you get, but the photographer still owns the copyright and has the legal monopoly on making more copies of those pictures. Surprise! Same applies to code.

Every software company is going to want to own the copyright to the code that its employees write for them, so *no* software company can accept the "default" way the law works. That is why *all* software companies that are well-managed will require *all* developers, at the very least, to sign an agreement that says, at the very least, that

- in exchange for receiving a salary,
- the developer agrees to "assign" (give) the copyright to the company.

This agreement can happen in the employment contract or in a separate "Proprietary Invention Assignment" contract. The way it is often expressed is by using the legal phrase [work for hire](), which means "we have decided that the copyright will be owned by the company, not the employee."

Now, we still haven't said anything about spare time work yet. Suppose, now, you have a little game company. Instead of making software, you knock out three or four clever games every few months. You can't invent all the games yourself. So you go out and hire a game designer to invent games. You are going to pay the game designer $6,000 a month to invent new games. Those games will be clever and novel. They are patentable. It is important to you, as a company, to own the *patents* on the games.

Your game designer works for a year and invents 7 games. At the end of the year, she sues you, claiming that she owns 4 of them, because those particular games were invented between 5pm and 9am, when she wasn't on duty.

Ooops. That's not what you meant. You wanted to pay her for *all* the games that she invents, and you recognize that the actual *process of invention* for which you are paying her may happen at any time… on weekdays, weekends, in the office, in the cubicle, at home, in the shower, climbing a mountain on vacation.

So before you hire this developer, you agree, "hey listen, I know that inventing happens all the time, and it's impossible to prove whether you invented something while you were sitting in the chair I supplied in the cubicle I supplied or not. I don't just want to buy your 9:00-5:00 inventions. I want them all, and I'm going to pay you a nice salary to get them all," and she agrees to that, so now you want to sign something that says that all her inventions belong to the company for as long as she is employed by the company.

This is where we are by default. This is the *standard* employment contract for developers, inventors, and researchers.

Even if a company decided, "oh gosh, we don't want to own the 5:00-9:00 inventions," they would soon get into trouble. Why? Because they might try to take an investment, and the investor would say, "prove to me that you're not going to get sued by some disgruntled ex-employee who claims to have invented the things that you're selling." The company wants to be able to pull out a list of all current and past employees, and show a contract from *every single one of them*assigning inventions to the company. This is expected as a part of due diligence in every single high tech financing, merger, and acquisition, so a software company that isn't careful about getting these assignments is going to have trouble getting financed, or merging, or being acquired, and that ONE GUY from 1998 who didn't sign the agreement is going to be a real jerk about signing it now, because he knows that he's personally holding up a $350,000,000 acquisition and he can demand a lot of money to sign.

So… every software company *tries* to own *everything* that its employees do. (They don't *necessarily* enforce it in cases of unrelated hobby projects, but on paper, they probably can.)

Software developers, as you can tell from this thread, found this situation to be upsetting. They always imagined that they should be able to sit in their own room at night on their own computer writing their own code for their own purposes and own the copyright and patents. So along came state legislators, in certain states (like California) but not others (not New York, for example). These state legislatures usually passed laws that said something like this:

Anything you do on your own time, with your own equipment, that is not related to your employer's line of work is yours, even if the contract you signed says otherwise.

Because this is the law of California, this particular clause is built into the standard Nolo contract and most of the standard contracts that California law firms give their software company clients, so programmers all over the country might well have this in their contract even if their state doesn't require it.

Let's look at that closely.

On your own time. Easy to determine, I imagine.

With your own equipment. Trivial to determine.

Not related to your employer's line of work. Um, wait. What's the definition of *related?* If my employer is Google, they do *everything.* They made a goddamn HOT AIR BALLOON with an internet router in it once. Are hot air balloons related? Obviously search engines, mail, web apps, and advertising are related to Google's line of work. Hmmm.

OK, what if my employer is a small company making software for the legal industry. Would software for the accounting industry be "related"?

I don't know. It's a big enough ambiguity that you could drive a truck through it. It's probably going to depend on a judge or jury.

The judge (or jury) is *likely* to be friendly to the poor employee against Big Bad Google, but you can't depend on it.

This ambiguity is meant to create enough of a chilling effect on the employee working in their spare time that for all intents and purposes it achieves the effect that the employer wants: the employee doesn't bother doing any side projects that might turn into a business some day, and the employer gets a nice, refreshed employee coming to work in the morning after spending the previous evening watching TV.

So… to answer the question. There is unlikely to be substantial difference between the contracts that you sign at various companies in the US working as a developer or in the law that applies. All of them need to purchase your copyright and patents without having to prove that they were generated "on the clock," so they will all try to do this, unless the company is being negligent and has not arranged for appropriate contracts to be in place, in which case, the company is probably being badly mismanaged and there's another reason not to work there.

The only difference is in the stance of management as to how hard they want to *enforce* their rights under these contracts. This can vary from:

- We love side projects. Have fun!
- We don't really like side projects. You should be thinking about things for us.
- We love side projects. We love them so much we want to own them and sell them!
- We are kinda indifferent. If you piss us off, we will look for ways to make you miserable. If you leave and start a competitive company or even a half-competitive company, we will use this contract to bring you to tears. BUT, if you don't piss us off, and serve us loyally, we'll look the other way when your iPhone app starts making $40,000 a month.

It may vary depending on whom you talk to, who is in power at any particular time, and whether or not you're sleeping with the boss. You're on your own, basically—the only way to gain independence is to be independent. Being an employee of a high tech company whose *product* is intellectual means that you have decided that you want to sell your intellectual output, and maybe that's OK, and maybe it's not, but it's a free choice. ■

This AI boom will also bust

By ROBIN HANSON

Imagine an innovation in pipes. If this innovation were general, something that made all kinds of pipes cheaper to build and maintain, the total benefits could be large, perhaps even comparable to the total amount we spend on pipes today. (Or even much larger.) And if most of the value of pipe use were in many small uses, then that is where most of these economic gains would be found.

In contrast, consider an innovation that only improved the very largest pipes. This innovation might, for example, cost a lot to use per meter of pipe, and so only make sense for the largest pipes. Such an innovation might make for very dramatic demonstrations, with huge vivid pipes, and so get media coverage. But the total economic gains here will probably be smaller; as most of pipe value is found in small pipes, gains to the few biggest pipes can only do so much.

Now consider my most viral tweet so far:

> *Good CS expert says: Most firms that think they want advanced AI/ML really just need linear regression on cleaned-up data.*

This got almost universal agreement from those who see such issues play out behind the scenes. And by analogy with the pipe innovation case, this fact tells us something about the potential near-term economic impact of recent innovations in Machine Learning. Let me explain.

Most firms have piles of data they aren't doing much with, and far more data that they could collect at a modest cost. Sometimes they use some of this data to predict a few things of interest. Sometimes this creates substantial business value. Most of this value is achieved, as usual, in the simplest applications, where simple prediction methods are applied to simple small datasets. And the total value achieved is only a small fraction of the world economy, at least as measured by income received by workers and firms who specialize in predicting from data.

Many obstacles limit such applications. For example, the value of better predictions for related decisions may be low, data may be in a form poorly suited to informing predictions, making good use of predictions might require larger reorganizations, and organizations that hold parts of the data may not want to lose control of that data. Available personnel may lack sufficient skills to apply the most effective approaches for data cleaning, merging, analysis, and application.

No doubt many errors are made in choices of when to analyze what data how much and by whom. Sometimes they will do too much prediction, and sometimes too little. When tech changes, orgs will sometimes wait too long to try new tech, and sometimes will not wait long enough for tech to mature. But in ordinary times, when the relevant technologies improve at steady known rates, we have no strong reason to expect these choices to be greatly wrong on average.

In the last few years, new "deep machine learning" prediction methods are "hot." In some widely publicized demonstrations, they seem to allow substantially more accurate predictions from data. Since they shine more when data is plentiful, and they need more skilled personnel, these methods are most promising for the largest prediction problems. Because of this new fashion, at many firms those who don't understand these issues well are pushing subordinates to seek local applications of these new methods. Those subordinates comply, at least in appearance, in part to help they and their organization appear more skilled.

One result of this new fashion is that a few big new applications are being explored, in places with enough data and potential prediction value to make them decent candidates. But another result is the one described in my tweet above: fashion-induced overuse of more expensive new methods on smaller problems to which they are poorly matched. We should expect this second result to produce a net loss on average. The size of this loss could be enough to outweigh all the gains from the few big new applications; after all, most value is usually achieved in many small problems.

But I don't want to draw a conclusion here about the net gain or loss. I instead want to consider the potential for this new prediction tech to have an overwhelming impact on the world economy. Some see this new fashion as just first swell of a tsunami that will soon swallow the world. For example, in 2013 Frey and Osborne famously estimated:

About 47 percent of total US employment is at risk .. to computerisation .. perhaps over the next decade or two.

If new prediction techs induced a change that big, they would be creating a value that is a substantial fraction of the world economy, and so consume a similar fraction of world income. If so, the prediction industry would in a short time become *vastly* larger than it is today. If today's fashion were the start of that vast growth, we should not only see an increase in prediction activity, we should also see an awe-inspiring rate of *success* within that activity. The application of these new methods should be enabling huge new revenue streams, across a very wide range of possible application areas. (*Added*: And the prospect of that should be increasing stock values in this area far more than we've seen.)

But I instead hear that within the areas where most prediction value lies, most attempts to apply this new tech actually produce *less* net value than would be achieved with old tech. I hear that prediction analysis tech is usually not the most important part the process, and that recently obsession with showing proficiency in this new analysis tech has led to neglect of the more important and basic issues of thinking carefully about what you might want to predict with what data, and then carefully cleaning and merging your data into a more useful form.

Yes, there must be exceptions, and some of those may be big. So a few big applications may enable big value. And self-driving cars seem a plausible candidate, a case where prediction is ready to give large value, high enough to justify using the most advanced prediction tech, and where lots of the right sort of data is available. But even if self-driving vehicles displace most drivers within a few decades, that rate of job automation wouldn't be out of the range of our historical record of job automation. So it wouldn't show that "this time is different." To be clearly out of that range, we'd need another ten jobs that big also displaced in the same period. And even that isn't enough to automate half of all jobs in two decades.

The bottom line here is that while some see this new prediction tech as like a new pipe tech that could improve all pipes, no matter their size, it is actually more like a tech only useful on very large pipes. Just as it would be a waste to force a pipe tech only useful for big pipes onto all pipes, it can be a waste to push advanced prediction tech onto typical prediction tasks. And the fact that this new tech is mainly only useful on rare big problems suggests that its total impact will be limited. It just isn't the sort of thing that can remake the world economy in two decades. To the extent that the current boom is based on such grand homes, this boom must soon bust. ■

Zero cost abstractions

By RUUD VAN ASSELDONK

If you read my blog, you can probably guess that I like Rust. I like Rust for a variety of reasons. Its error model, for example. But today I want to highlight a different reason: zero-cost abstractions.

Recently I had the opportunity to do some optimisation work on Claxon, my FLAC decoder. In the process I found a piece of code which in my opinion demonstrates the power of Rust very well, and I would like to share that with you here:

```
for i in 12..buffer.len() {
    let prediction = coefficients.iter()
                                 .zip(&buffer[i - 12..i])
                                 .map(|(&c, &s)| c * s as i64)
                                 .sum::<i64>() >> qlp_shift;
    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

For context, the following variables are in scope:

```
buffer: &mut [i32];      // A mutable array slice.
coefficients: [i64; 12];  // A fixed-size array of 12 elements.
qlp_shift: i16;
```

The snippet is part of a function that restores sample values from residues. This is something that happens a lot during decoding, and this particular loop makes up roughly 20% of the total decoding time. It had better be efficient.

Ingredients

The snippet computes a fixed-point arithmetic inner product between the coefficients and a window that slides over the buffer. This value is the prediction for a sample. After adding the residue to the prediction, the window is advanced. An inner product can be neatly expressed by zipping the coefficients with the window, mapping multiplication over the pairs, and then taking the sum. I would call this an abstraction: it moves the emphasis away from how the value is computed, focusing on a declarative specification instead.

The snippet is short and clear — in my opinion — but there is quite a bit going on behind the scenes. Let's break it down.

- `12..buffer.len()` constructs a `Range` structure with an upper and lower bound. Iterating over it with a for loop implicitly creates an iterator, however in the case of `Range` that iterator is the structure itself.
- `coefficients().iter()` constructs a slice iterator, and the call to `zip` implicitly constructs an iterator for the `buffer` slice as well.
- `zip` and `map` both wrap their input iterators in a new iterator structure.
- A closure is being passed to `map`. The closure does not capture anything from its environment in this case.
- `sum` repeatedly calls `next()` on its input iterator, pattern matches on the result, and adds up the values until the iterator is exhausted.
- Indexing into and slicing `buffer` will panic if an index is out of bounds, as Rust does not allow reading past the end of an array.

It appears that these high-level constructs come at a price. Many intermediate structures are created which would not be present in a hand-written inner product. Fortunately these structures are not allocated on the heap, as they would likely be in a language like Java or Python. Iterators also introduce extra control flow: `zip` will terminate after one of its inner iterators is exhausted, so in principle it has to branch twice on every iteration. And of course iterating itself involves a call to `next()` on every iteration. Are we trading performance for convenience here?

Generated code

The only proper way to reason about the cost of these abstractions is to inspect the generated machine code. Claxon comes with a `decode` example program, which I compiled in release mode with Rustc 1.13 stable. Let's take a look at the result:

```
10c00:
movslq %r14d,%r11
movslq -0x2c(%r8,%rdi,4),%rsi
imul   %r10,%rsi
movslq -0x30(%r8,%rdi,4),%r14
imul   %rbp,%r14
add    %rsi,%r14
movslq -0x28(%r8,%rdi,4),%rsi
imul   %rdx,%rsi
add    %rsi,%r14
movslq -0x24(%r8,%rdi,4),%rsi
imul   %rax,%rsi
add    %rsi,%r14
movslq -0x20(%r8,%rdi,4),%rsi
imul   %rbx,%rsi
add    %rsi,%r14
movslq -0x1c(%r8,%rdi,4),%rsi
imul   %r15,%rsi
add    %rsi,%r14
movslq -0x18(%r8,%rdi,4),%rsi
imul   %r13,%rsi
add    %rsi,%r14
movslq -0x14(%r8,%rdi,4),%rsi
imul   %r12,%rsi
add    %rsi,%r14
movslq -0x10(%r8,%rdi,4),%rsi
imul   0x8(%rsp),%rsi
add    %rsi,%r14
movslq -0xc(%r8,%rdi,4),%rsi
imul   0x18(%rsp),%rsi
add    %rsi,%r14
movslq -0x8(%r8,%rdi,4),%rsi
imul   0x20(%rsp),%rsi
add    %rsi,%r14
imul   0x10(%rsp),%r11
add    %r11,%r14
sar    %cl,%r14
add    (%r8,%rdi,4),%r14d
mov    %r14d,(%r8,%rdi,4)
inc    %rdi
cmp    %r9,%rdi
jb     10c00 <claxon::subframe::predict_lpc::h6c02f07b190820c0+0x2b0>
```

All overhead is gone **completely**. What happened here? First of all, no loop is used to compute the inner product. The input slices have a fixed size of 12 elements, and despite the use of iterators, the compiler was able to unroll everything here. The element-wise computations are efficient too. There are 12 `movslq`s which load a value from the buffer and simultaneously widen it. There are 12 multiplications and 12 additions, 11 for the inner product, and one to add the delta after arithmetic shifting `(sar)` the sum right. Note that the coefficients are not even loaded inside the loop, they are kept in registers at all times. After the sample has been computed, it is stored simply with a `mov`. All bounds checks have been elided. The final three instructions handle control flow of the loop. Not a single instruction is redundant here, and I could not have written this better myself.

I don't want to end this post without at least touching briefly upon vectorisation. It might look like the compiler missed an opportunity for vectorisation here, but I do not think that this is the case. For various reasons the above snippet is not as obvious to vectorise as it might seem at first sight. But in any case, a missed opportunity for vectorisation is just that: a missed opportunity. It is not abstraction overhead.

Conclusion

In this post I've shown a small snippet of code that uses high-level constructs such as closures and iterator combinators, yet the code compiles down to the same instructions that a hand-written C program would compile to. Rust lives up to its promise: abstractions are truly zero-cost. ■

I don't belong in tech

Trying to find my place in the place I love, and constantly failing

By SARON YITBAREK

It was dark and cold that night I stomped down Broadway, talking to my then-boyfriend-now-husband about my feelings. I am always talking about my feelings, and he is always listening. He "mhm"s at the right places and doesn't interrupt and sometimes says good things at the end. Sometimes he says wrong things, and then I have to explain why those things are wrong, taking us down an emotional tangent that is frustrating and exhausting, but he's trying to be helpful, I tell myself and breathe. Bless his little heart.

But tonight, he lets me talk. And I do, filling the minutes with long, twisting sentences that make sense to me, but as they tumble out, I'm not sure that they do, so I pause and I blurt, "I'm just not a white man!" Or something like that. This was years ago, so who knows what really happened. I may not have been on Broadway at all. But that's where the anger ended, in not being white or a man or coding since I was two or some combination of the three. This wasn't going to work. Coding wasn't going to work. I didn't belong.

Fast-forward three years. I'd choked down my feelings and learned to code and built things and knew stuff that even my then-boyfriend-now-husband didn't know. We sat on our couch one evening while I explained how AJAX worked. He leaned back and I leaned in, excited and trembling at the edge of my seat. I heard the words coming out of my mouth, watched them float in the air between us, blooming with buzzwords and jargon and pride and I burst into tears. I covered my face with my hands, hunched over and shook. I couldn't believe I understood the words I was saying. This was going to work. Coding was going to work. Maybe I did belong.

The cracks in that newly laid confidence would soon come, but not for reasons I may have lead you to believe. I apologize if you assumed this was a story about a difference rooted in race and gender, because it is not. That's not where we are going. This is about a difference of values, beliefs, perspectives.

I wanted so badly to think like a programmer, which implies that the way I think is wrong. It needed fixing in many ways. This observation is frustratingly fuzzy, cloudy, unfocused, but I've squeezed it hard enough to make raindrops, something I can taste and feel, and I shall give you three.

I am not solution-oriented. I don't see a problem and get giddy at the idea of solving it, patching it up and sending it on its merry way. I want to poke it and ask it questions. Where did it come from, what is it doing, what's its story? I want to take it to tea and hear about its life and understand it to its core. And if, at that point, I've come to a holistic understanding and am able to solve the problem, by all means, let the problem-solving commence! But my instinct is never to solve, but to understand.

This is the part where you tell me that this is a great asset in a programmer! That all programmers would be much better off if they took the time to understand before diving in! My thinking isn't broken at all, you say, it's super awesome!

That's cute. And truly, I appreciate your defense of my broken brain. But no matter what Medium blog posts tell you how crucial it is to understand the problem before coding its solution, this is, at best, an annoying part of an average developer's job, and, more likely, a distant idea that is happily ignored. Developers solve problems. It is the problem-solving, not the problem-understanding, that gets you high.

Hm. Maybe this isn't going to work.

I am not comfortable making half-ass shit. Once in a while I look up the famous quote by Reid Hoffman, co-founder of LinkedIn, who said, "If you aren't embarrassed by the first version of your product, you shipped too late." I say it to myself. I say it again. I let it sit and turn it over and convince myself this isn't insane. I understand this concept at an intellectual level. I get the value of the MVP (minimum viable product) and was excited to learn the pseudo-scientific process of the lean methodology. The quote is a punchy way of encouraging product creators to start small and test an idea before investing loads of money and time in an expensive mistake.

And this advice is great! You *should* start small and test and learn. But the way this advice manifests itself is often in writing shitty shit that makes shitty shit products, and leaving it in its shitty shit way. It's the shrug that accompanies the mindless defense, "But it works." It produces a mentality of doing the absolute bare minimum, not because it's what's best for the product or your team, but because, why bother to do more? It works! It condemns everything I've learned and loved about craftsmanship and quality and just plain giving a fuck. There are no As here, there is only pass and fail. Maybe coding wasn't going to work.

This is the part where you tell me that there *is* such a thing as beautiful code! There are talks that preach the value of well written code, books filled with advice on how to hone your craft, podcast interviews of developers raging against poorly written programs. My value of quality is wonderful, you say, do not fix it, you shout, keep going, you plead!

That's cute. But no matter how many conference talks you've tweeted about praising code as craft, open up your company's production-level app right now and tell me how much of that has made its way to your product. Don't worry, I'll wait. Because in the real world of death marches, limited runway, and just plain old pressure from the higher ups, quality and care are a dream: sweet, distant, and rarely realized.

But perhaps the biggest way that my brain is broken is less about code, and more about the tech industry as a whole. If you're thinking to yourself, "But everyone uses tech so everything is the tech industry," please sit tight while I take a moment to roll my eyes. … Ok, I'm back. For our purposes, let's define "tech industry" as companies and professionals who view code as a core part of their business and their self-understanding, both internally and externally.

When I was at NPR years ago, I did a story on public education in California. I don't remember the angle, but I remember looking up a stat to use in the script. I used that stat in a few places, and after fact-checking, I realized there was an updated number available. I went back and changed the references to the new number, relieved that I'd caught this mistake before handing over my script to the host. But I missed one. I heard it over the speakers when Michelle Martin, the host, read it out loud during the interview, and my heart stopped. I knew it was my duty to report it, so I went up to my editor and told her. She didn't say anything, but I could feel her disappointment in me. I melted into a pool of shame.

But here's the thing. No one will ever remember that number. No one remembers it now, and I'm sure no one noticed it when it happened. But *I* knew it happened, that it was an easily preventable mistake, and, in journalism, being wrong in that way is absolutely unacceptable. So imagine my surprise when I first heard of "fail fast and break things," one of the famous tech mantras for product creation. Imagine my shock to find out that being wrong is not reprimanded, but, at times, encouraged. Imagine my confusion stepping into a world where people are told to "just try it and see." I tell myself over and over that this is different, that this is good, that public experimentation is not a holy sin. I've managed to convince myself, when I'm not busy quieting a nauseous tummy tormented by public broken attempts and shameful failures. But here, I will admit defeat. Being wrong in software is fundamentally different from being wrong in reporting. Except when it's not.

When I use your product, I'm trusting you. I believe you when you tell me that clicking that button will create my profile, that I am indeed submitting an email by hitting enter, that I will see my mom's message when I click on her little, round face. My belief in you is delicate and deep. Do not take my trust for granted. Do not take advantage of me.

We are in a relationship, you and I. Distant and faceless, yes, but a relationship nonetheless. I give and you take and you give and I take, and I believe your words, your lines, your interfaces. It should be precious. It should be handled with care, but the carelessness I see in tech is unsettling. The willful ignorance, the rejection of our relationship, hurts.

It might come big, like playing with my emotions by purposefully filling my feed with sad or happy content, just to see how I respond.

It might come small, like your claim of being the number one this-and-that in your this-and-that field, according to … no one. You are so proud of your accomplishments and so comfortable in your grandeur that you forget to be honest with me.

Sometimes it comes deep, like spending months together trying to solve a problem you promised me you could solve to later find out that you got it all wrong, you made it all up, you have no idea what you're doing. You brag about this in your interviews and inevitable autobiography. For some strange reason, you wear this ignorance as a badge of honor. You failed fast and broke my heart.

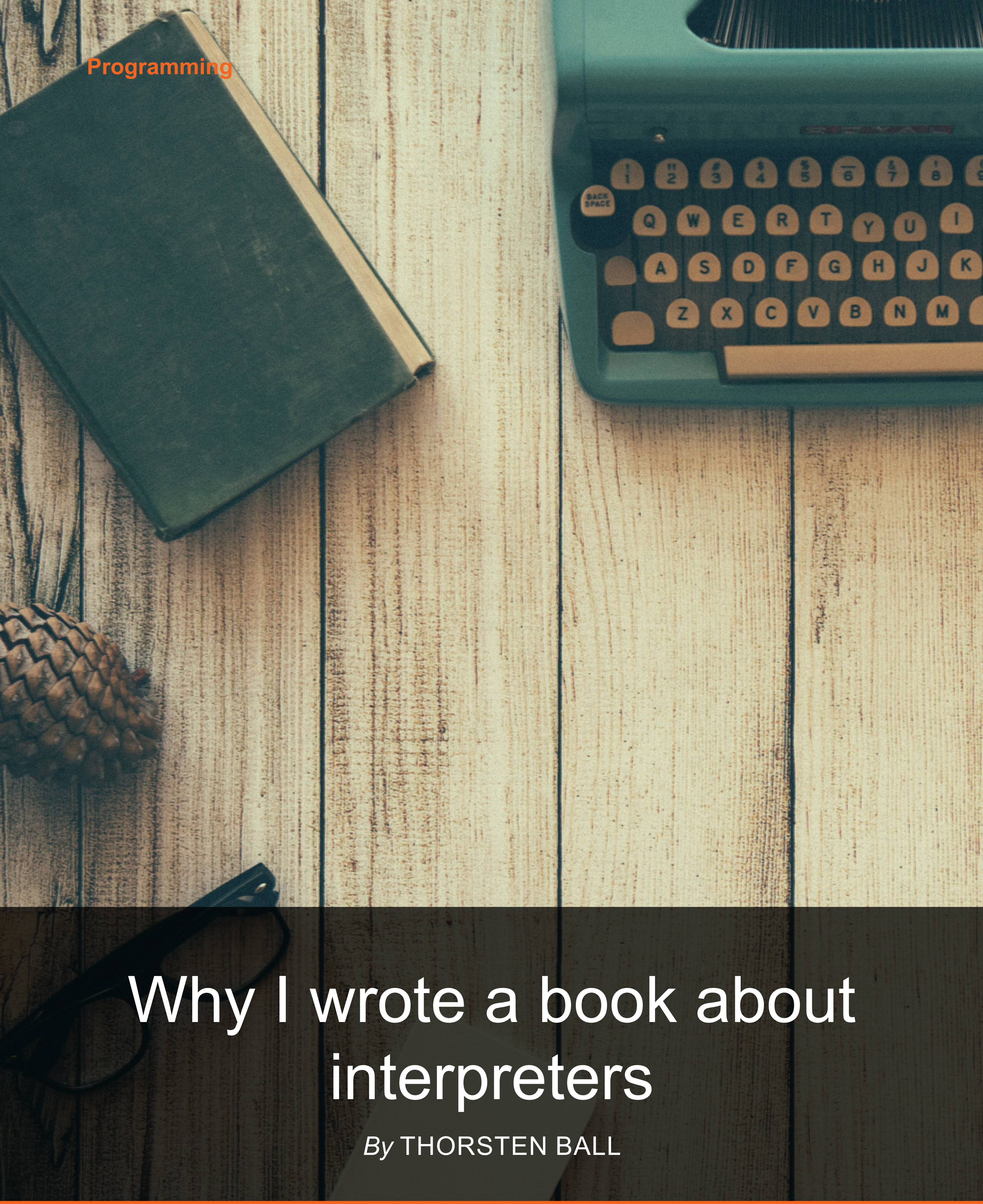
But you will never see it that way. You're too excited. I feel you whisper *make the world a better place* as you drift to sleep, so obsessed with changing it that you forget that the world is made up of little people like me.

You are experimenting, trying new things, and for this, you are great and lean. But sometimes, you forget that I'm at the center of your experiments. Sometimes, you forget me.

I take these relationships seriously. So seriously that often I'm immobilized and overwhelmed. And in those moments, you push products I'm too uncomfortable to push and you win. You get there first, making waves while I sit in last place and watch. So I choke down my values and discomfort and attempt a push of my own, amid the internal screams that this is wrong and irresponsible and how dare I. I don't get very far. My feeble, half-hearted steps cannot compete with your bold, proud strides. So I cower back to my corner with my broken brain and peep at your success through the leaves.

I do not belong. My values are not valued. My thinking is strange and foreign. My world view has no place here. It is not that I am better, it is that I am different, and my difference feels incompatible with yours, dear tech. So I will mark my corner, a small plot of land and stand firmly here, trying to understand you and reconcile these conflicting differences.

Maybe I will change. Maybe you'll surprise me. Maybe, one day, I'll belong. ■



Why I wrote a book about interpreters

By THORSTEN BALL

Last week I've self-published my first book called "Writing An Interpreter In Go", which you can get at interpreterbook.com. I want to tell you a little bit about why I chose to write this particular book.

Sometimes I jokingly call the summer of 2015 my "Summer Of Lisp". But, honestly, I'm only half joking when I say this. It really was a great and Lispy summer programming-wise: I was working through the final chapters of Structure And Interpretation Of Computer Programs (SICP), which I began studying at the beginning of that year, was totally fascinated by Lisp, enamored by Scheme and also starting to learn Clojure by working through the fantastic The Joy Of Clojure.

SICP had an immense impact on me. It's a wonderful book, full of elegant code and ideas; it hearkens "to a programming life that if true, would be an absolute blast to live in". Especially the fourth chapter made a lasting impression. In this chapter, Abelson and Sussmann show the reader how to implement the so called "meta-circular evaluator" - a Lisp interpreter in Lisp. "Mesmerized" is probably the word I'd use to describe myself while reading this chapter.

The code for the meta-circular evaluator is elegant and simple. Around 400 lines of Scheme, stripped down to the essentials and doing exactly what they are supposed to. It's a beautiful piece of software. I asked a friend to design a poster for me, containing only the source code for the meta-circular interpreter, beautifully formatted. That poster hung next to my office desk for over a year.

But soon I discovered why it's only 400 lines. The code presented in the book skips the implementation of an entire component - the parser. *Huh. But how does a parser work then?* I was stumped. I really wanted to know how that parser works. And I almost never want to *skip anything* I don't know yet. I really want to know how things work, at least in a rough sense. Black boxes and skipping things always leave me wanting to dig deeper.

In that same summer I also read Steve Yegge's "Rich Programmer Food", in which he argues what a worthwhile goal it is to learn about and to understand compilers. Let me quote my favorite passage:

That's why you need to learn how [compilers] work. That's why you, yes you personally, need to write one.

[…]

You'll be able to fix that dang syntax highlighting.

You'll be able to write that doc extractor.

You'll be able to fix the broken indentation in Eclipse.

You won't have to wait for your tools to catch up.

You might even stop bragging about how smart your tools are, how amazing it is that they can understand your code […]

You'll be able to jump in and help fix all those problems with your favorite language.

That blog post flipped a switch. Determined as if there was some kind of weird challenge I said to a friend of mine: "I'm going to write a compiler". I believe, I was gazing into the distance while saying this. "Alright", he said rather unimpressed, "do it."

Without having taken a compiler course in college or even having a computer science degree I set out to write a compiler. The first goal, I determined, is to get a foot in the door and write an interpreter.

Interpreters are closely related to compilers, but easier to understand and to build for beginners. But most importantly, this time there would be no skipping of anything. *This interpreter will be built from scratch!*

What I found was that a lot of resources for interpreters or compilers are either incredibly heavy on theory or barely scratching the surface. It's either the dragon book or a blog post about a 50 line Lisp interpreter. The complete theory with code in the appendix or an introduction and overview with black boxes.

Every piece of writing helped though. Slowly but surely I was completing work on my interpreter. The tiny tutorials, the slightly longer blog posts and the heavy compiler books - I could find something useful in all of them.

Nevertheless I was getting frustrated. *There needs to be a book, that* … One day, I said to the same friend, who earlier so enthusiastically encouraged me to write a compiler:

"You know what… I'd love to write a book about interpreters. A book that shows you everything you need to know to build an interpreter from scratch, including your own lexer, your own parser and your own evaluation step. No skipping of anything!"

Somehow this turned into me giving myself a motivational speech.

"And with tests too!", I continued, "Yeah! Code and tests front and center! Not like in these other books, where the code is an unreadable mess that you can't get to compile or run on your system. And you don't need to be well versed in mathematical notation either! It should be a book any programmer can read and understand."

It's entirely possible that I was banging my fist on the table at this point. Calmly, my friend said: "Sounds like a good idea. Do it."

And here we are, 11 months later, and "Writing An Interpreter In Go" is available to the public. It has around 200 pages and presents the complete and working interpreter for the Monkey programming language, including the lexer, the parser, the evaluator and also including tests. No black boxes, no 3rd party tools and no skipping of anything. Nearly every page contains a piece of code. I'm really proud of this book. ■

Taco bell programming

By TED DZIUBA

Every item on the menu at Taco Bell is just a different configuration of roughly eight ingredients. With this simple periodic table of meat and produce, the company pulled down $1.9 billion last year.

The more I write code and design systems, the more I understand that many times, you can achieve the desired functionality simply with clever reconfigurations of the basic Unix tool set. After all, *functionality is an asset, but code is a liability*. This is the opposite of a trend of nonsense called DevOps, where system administrators start writing unit tests and other things to help the developers warm up to them - Taco Bell Programming is about developers knowing enough about Ops (and Unix in general) so that they don't overthink things, and arrive at simple, scalable solutions.

Here's a concrete example: suppose you have millions of web pages that you want to download and save to disk for later processing. How do you do it? The cool-kids answer is to write a distributed crawler in Clojure and run it on EC2, handing out jobs with a message queue like SQS or ZeroMQ.

The Taco Bell answer? `xargs and wget`. In the rare case that you saturate the network connection, add some `split` and `rsync`. A "distributed crawler" is really only like 10 lines of shell script.

Moving on, once you have these millions of pages (or even tens of millions), how do you process them? Surely, Hadoop MapReduce is necessary, after all, that's what Google uses to parse the web, right?

Pfft, fuck that noise:

```
find crawl_dir/ -type f -print0 | xargs -n1 -0 -P32 ./process
```

32 concurrent parallel parsing processes and zero bullshit to manage. Requirement satisfied.

Every time you write code or introduce third-party services, you are introducing the possibility of failure into your system. I have far more faith in `xargs` than I do in Hadoop. Hell, I trust `xargs` more than I trust myself to write a simple multithreaded processor. I trust syslog to handle asynchronous message recording far more than I trust a message queue service.

Taco Bell programming is one of the steps on the path to Unix Zen. This is a path that I am personally just beginning, but it's already starting to pay dividends. To really get into it, you need to throw away a lot of your ideas about how systems are designed: I made most of a SOAP server using static files and Apache's `mod_rewrite`. I could have done the whole thing Taco Bell style if I had only manned up and broken out `sed`, but I pussied out and wrote some Python.

If you don't want to think of it from a Zen perspective, be capitalist: you are writing software to put food on the table. You can minimize risk by using the well-proven tool set, or you can step into the land of the unknown. It may not get you invited to speak at conferences, but it will get the job done, and help keep your pager from going off at night. ■

Using Rust for 'scripting'

By CHRIS KRYCHO

I. Using Rust instead of Python

A friend asked me today about writing a little script to do a simple conversion of the names of some files in a nested set of directories. Everything with one file extension needed to get another file extension. After asking if it was the kind of thing where he had time to and/or wanted to learn how to do it himself (always important when someone has expressed that interest more generally), I said, "Why don't I do this in Rust?"

Now, given the description, you might think, *Wouldn't it make more sense to do that in Python or Perl or even just a shell script?* And the answer would be: it depends—on what the target operating system is, for example, and what the person's current setup is. I knew, for example, that my friend is running Windows, which means he doesn't have Python or Perl installed. I'm not a huge fan of either batch scripts or PowerShell (and I don't know either of them all that well, either).

I could have asked him to install Python. But, on reflection, I thought: *Why would I do that? I can write this in Rust.*

Writing it in Rust means I can compile it and hand it to him, and he can run it. And that's it. As wonderful as they are, the fact that languages like Python, Perl, Ruby, JavaScript, etc. require having the runtime bundled up with them makes just shipping a tool a lot harder—*especially* on systems which aren't a Unix derivative and don't have them installed by default. (Yes, I know that *mostly* means Windows, but it doesn't *solely* mean Windows. And, more importantly: the vast majority of the desktop-type computers in the world *still run Windows*. So that's a big reason all by itself.)

So there's the justification for shipping a compiled binary. Why Rust specifically? Well, because I'm a fanboy. (But I'm a fanboy because Rust often gives you roughly the feel of using a high-level language like Python, but lets you ship standalone binaries. The same is true of a variety of other languages, too, like Haskell; but Rust is the one I know and like right now.)

I'm not actually advocating that everyone stop using shell scripts for this kind of thing. I'm simply noting that it's possible (and sometimes even nice) to be able to do this kind of thing in Rust, cross-compile it, and just ship it. And hey, types are nice when you're trying to do more sophisticated things than I'm doing here! Also, for those worried about running untrusted binaries: I handed my friend the code, and would happily teach him how to build it.

II. Building a simple "script"

Building a "script"-style tool in Rust is pretty easy, gladly. I'll walk through exactly what I did to create this "script"-like tool for my friend. My goal here was to rename every file in a directory from `*.cha` to `*.txt`.

1. Install Rust.
2. Create a new binary:

```
cargo new --bin rename-it
```

3. Add the dependencies to the Cargo.toml file. I used the glob crate for finding all the `.cha` files and the clap crate for argument parsing.

```
[package]
name = "rename-it"
```

```
version = "0.1.0"
authors = ["Chris Krycho <chris@chriskrycho.com>"]

[dependencies]
clap = "2.15.0"
glob = "0.2"
```

4. Add the actual implementation to the `main.rs` file (iterating till you get it the way you want, of course).

```
extern crate clap;
extern crate glob;

use glob::glob;
use std::fs;

use clap::{Arg, App, AppSettings};


fn main() {
  let path_arg_name = "path";
  let args = App::new("cha-to-txt")
    .about("Rename .cha to .txt")
    .setting(AppSettings::ArgRequiredElseHelp)
    .arg(Arg::with_name(path_arg_name)
      .help("path to the top directory with .cha files"))
    .get_matches();

  let path = args.value_of(path_arg_name)
    .expect("You didn't supply a path");
  let search = String::from(path) + "/**/*.cha";
  let paths = glob(&search)
    .expect("Could not find paths in glob")
    .map(|p| p.expect("Bad individual path in glob"));

  for path in paths {
    match fs::rename(&path, &path.with_extension("txt")) {
      Ok(_) => (),
      Err(reason) => panic!("{}", reason),
    };
  }
}
```

5. Compile it.

```
cargo build --release
```

6. Copy the executable to hand to a friend.

In my case, I actually added in the step of *recompiling* it on Windows after doing all the development on macOS. This is one of the real pieces of magic with Rust: you can *easily* write cross-platform code. The combination of Cargo and native-compiled-code makes it super easy to write this kind of thing—and, honestly, easier to do so in a cross-platform way than it would be with a traditional scripting language.[1]

But what's really delightful is that we can do better. I don't even need to install Rust on Windows to compile a Rust binary for Windows.

[1] Again: you can do similar with Haskell or OCaml or a number of other languages. And those are great options; they are in some ways easier than Rust—but Cargo is really magical for this.

III. Cross-compiling to Windows from macOS

Once again, let's do this step by step. Three notes: First, I got pretty much everything other than the first and last steps here from WindowsBunny on the #rust IRC channel. (If you've never hopped into #rust, you should: it's amazing.) Second, you'll need a Windows installation to make this work, as you'll need some libraries. (That's a pain, but it's a one-time pain.) Third, this is the setup for doing in on macOS Sierra; steps may look a little different on an earlier version of macOS or on Linux.

1. Install the Windows compilation target with `rustup`.

```
rustup target add x86_64-pc-windows-msvc
```

2. Install the required linker (lld) by way of installing the LLVM toolchain.

```
brew install llvm
```

3. Create a symlink somewhere on your `PATH` to the newly installed linker, specifically with the name `link.exe`. I have `~/bin` on my `PATH` for just this kind of thing, so I can do that like so:

```
ln -s /usr/local/opt/llvm/bin/lld-link ~/bin/link.exe
```

   (We have to do this because the Rust compiler specifically goes looking for `link.exe` on non-Windows targets.)

4. Copy the target files for the Windows build to link against. Those are in these directories, where `<something>` will be a number like `10586.0` or similar (you should pick the highest one if there is more than one):

```
C:\Program Files\Windows Kits\10\Lib\10.0.<something>\ucrt\x64
C:\Program Files\Windows Kits\10\Lib\10.0.<something>\um\x64
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\lib\amd64
```

   Note that if you don't already have MSVC installed, you'll need to install it. If you don't have Visual Studio installed on a Windows machine *at all*, you can do that by using the links here. Otherwise, on Windows, go to **Add/Remove Programs** and opting to Modify the Visual Studio installation. There, you can choose to add the C++ tools to the installation.

   Note also that if you're building for 32-bit Windows you'll want to grab *those* libraries instead of the 64-bit libraries.

5. Set the `LIB` environment variable to include those paths and build the program. Let's say you put them in something like `/Users/chris/lib/windows` (which is where I put mine). Your Cargo invocation will look like this:

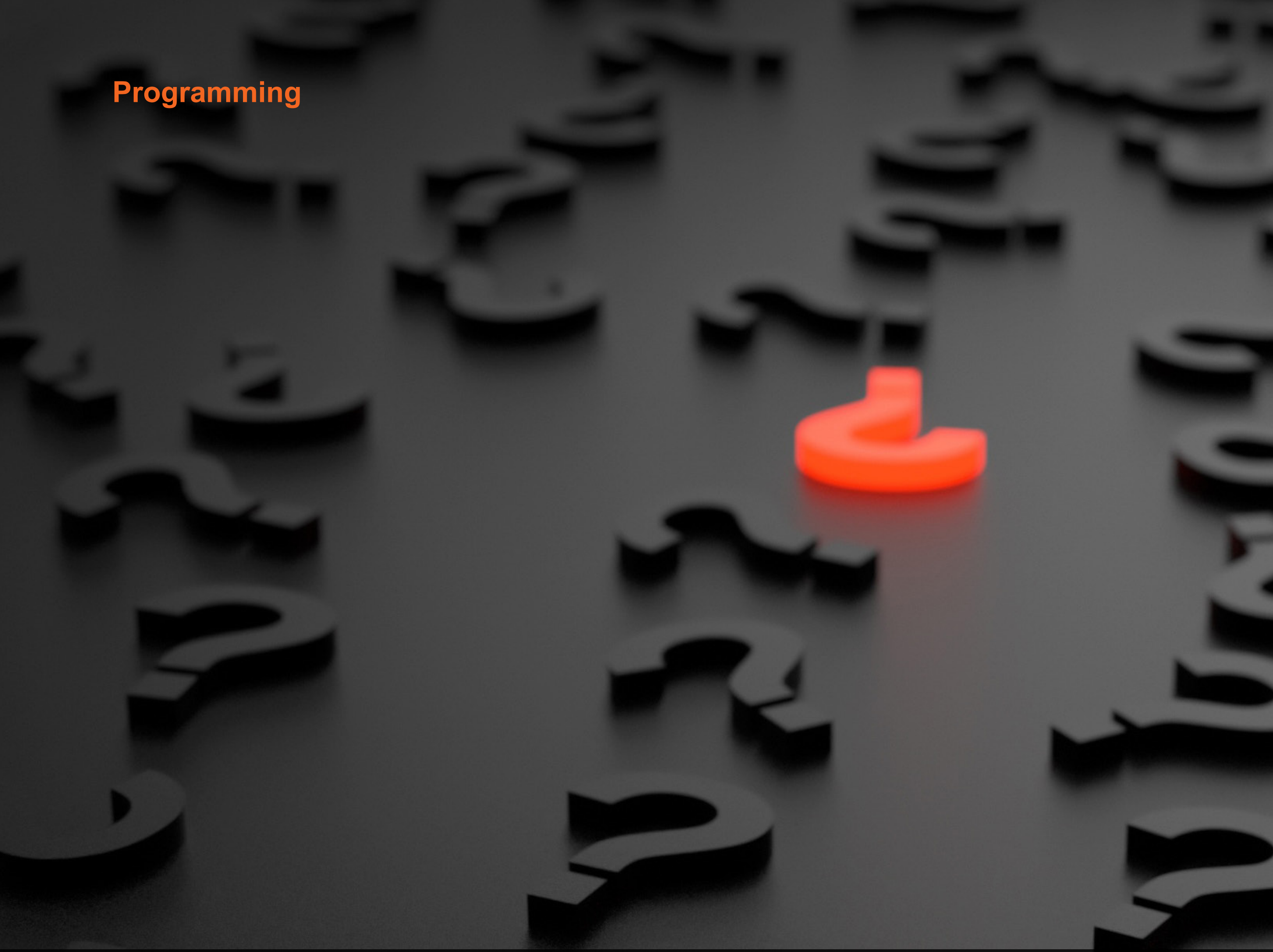
```
env LIB="/Users/chris/lib/windows/ucrt/x64/;/Users/chris/lib/windows/um/x64/;/Users/chris/lib/
windows/VC_lib/amd64/" \
cargo build --release --target=x86_64-pc-windows-msvc
```

   Note that the final `/` on each path and the enclosing quotation marks are all important!

6. Copy the binary to hand to a friend, without ever having had to leave your Mac.

To be sure, there was a little extra work involved in getting cross-compilation set up. (This is the kind of thing I'd love to see further automated with `rustup` in 2017!) But what we have at the end is pretty magical. Now we can just compile cross-platform code and hand it to our friends.

Given that, I expect not to be using Python for these kinds of tools much going forward. ■



Do you really want a single-page application framework?

First, sell your soul to the framework maintainer…

By MICHAEL S. MIKOWSKI

What's wrong with an SPA Framework?

I author and maintain quite a bit of software. Much of my code is available in commercial applications (SnapLogic, Qualaroo), OSS libraries on Github and NPM, or demonstration applications like Typebomb. I am the co-author of the book that Dr. Dobb's Journal named as one of the best developer books of 2014 and called it the "Master Handbook" for SPAs. Yet despite numerous requests I have not published an **SPA framework**.

Why? Because I don't want to lock developers into a platform that must be, by definition, opinionated and limited. An SPA framework provides safety at the cost of an investment into their platform and a loss of flexibility and control. They can hinder our ability to innovate and build quality software. For these reasons, I advocate developers focus on JavaScript language mastery and SPA architecture rather than the promise of a **silver bullet** SPA framework.

Before we proceed please note that everything has its place. SPA frameworks can be useful to new or casual developers who don't know or care how an SPA works. But before you hitch your star to Ext, Angular, Ember, Aurelia, Meteor, Backbone, Knockout, Vue, or Mercury or dozens of others, please consider that every shiny new tool has its cost. Here are some precautions you might want to consider before you dive in.

Frameworks and the inversion of control

When we use an SPA framework, the quality and capabilities of our application are strongly limited by it. This inversion of control is a major impediment in building a a nimble, flexible, testable, and maintainable application that can stand the test of time.

When we use a sound architecture and libraries instead, however, we can swap libraries out when they are updated or better one becomes available. **Or we can decide not to change a thing** if an update doesn't suit our needs. We can mix and match the **best-for-our-purpose** libraries instead of using a framework's mishmash of solutions of varying quality. Why, for example, should one bother *overriding* the mediocre mechanisms of a framework when just removing the framework can results in simpler and easier to maintain code?

Does this mean we need to write everything from scratch?

Absolutely not. jQuery and other best-in-class libraries can provide a more capable and complete foundation for building a modern SPA compared to many frameworks. We can start with a simple and clean SPA architecture, p10 as detailed in Single page web applications, JavaScript end-to-end (also available directly from Manning), and then add libraries that are best suited to our application. We can **leverage** jQuery's maturity, performance, excellent tools, and vast ecosystem instead of **competing** with it. We also avoid ceding control of our application to an SPA framework.

Frameworks and complexity === insanely long cycle times

SPA frameworks these days are approaching or exceeding the complexity ofJavaScript-to-X compilers with sadly similar results. If we want to go all-in, we can get a "two-fer" by selecting Angular 2 + TypeScript. That way we can shoot ourselves in **both** feet instead of one and greatly increase our cycle times. Who needs short cycle times anyway?

Well, actually, **we** do. If our overhead for producing working development code is greater than say, oh, 5 seconds, then somebody out there is definitely kicking our ass on cycle times. We can fail 300 times an hour with a 5 second cycle time. In other words, if it takes 200 failures before success, we will need a minimum of 40 minutes to resolve an issue. However, if our cycle time is 5 minutes because of the multi-compile overhead of an SPA framework, our minimum time before success will be 72 *times* longer, or over two full work-days. You make the call.

Long cycle times not only kill productivity, but they also stifle innovation because only so many solutions can be tried within any given period of time. To paraphrase Thomas Edison, the key to innovation is to learning how to fail really fast.

Frameworks DSLs aren't portable

SPA frameworks DSLs **aren't** the most portable of life skills, just like JavaScript-to-X compilers DSLs. How many developers have moved from GWT to YUI to Dojo to Ext to Backbone to Closure to Knockout to Knockback to Ember to Angular? How about sprinkling some Bootstrap, Sass, TypeScript, Coffee-Scipt, Cappuccino, HAML, and a few more compile steps in there just for the fun of it? Maybe one is better at learning HTML5, CSS3, JS really well and taking all the noise about these "silver bullets" with a grain of salt. See The fog of SPA

Been there, got the T-shirt, bombed the airport

I once used a framework (not my choice) and had to wait months for a new version to support a desired feature. Once the framework was updated, I discovered excruciating pain of trying to find and fix all the regressions. It wasn't easy, of course, because (a) some of the most dastardly bugs were within the framework itself, and (b) frameworks tend to intermingle display and business logic, so testing was tedious and difficult. Which, of course, brings us to our next subject…

What about testing?

Testing a Single Page Framework Web Applications (SPFWA?) often requires an **additional** framework for testing the simplest of logic. Selenium, ZombieJS, and other intricate solutions are often employed to test the most basic model logic that *shouldn't require display testing at all.* Yet there it is. Complexity breeds complexity.

When we use a sound architecture and libraries instead, however, we can easily decouple display and business logic so we can regression test our application in less than a second. My commit hook for the above application eventually ran the full regression suite *and* JSLint for all changed files in less than 4 seconds with over 600 assertions and ~95% coverage. We could refactor and reorganize the code *at-will* because only the most obscure bugs could sneak past the regression tests. The architecture and method of testing scales very well on larger projects too.

What are your preferred libraries

Updated 2016-11-20. Here are libraries we recommend:

Capability	Library	Notes
DOM + Util	jQuery	A powerful, stable, tight library
AJAX	jQuery	… but prefer WebSockets, see below
Client Data	TaffyDB	A powerful and flexible SQL-like client data management tool
DynamicCSS	PowerCSS	Insanely fast and efficient JS-CSS engine
Linting	JSLint	Avoid stupid mistakes with a commit hook
Events, promises	Global Events	Use the same event and promise methods for both logical and browser events
Routing	uriAnchor	A jQuery plugin for robust routing that includes support for dependent and independent query arguments
SVG	D3	Easy graphs and charts
	SVG	Low-level jQuery plugin
Templates	Dust	Uses a powerful template DSL that minimizes the temptation to intermingle business and display logic
Testing	Nodeunit-b	Create a lightening fast regression test suite and use it as a commit hook
Touch	Unified events	Unified desktop and touch events
WebSockets	Socket io	The WebSockets protocol is faster and more flexible than AJAX for most applications. Consider using pure websockets client with a websocket server on a NodeJs with modern browsers (IE10+)

But I want the comfort of a Framework [Updated 2016-11-20]

The NPM package hi_score is moving along nicely. It includes all the tools needed to deploy a modern, well packaged SPA for production. It includes architecture diagrams, example code, code compression and obsfucation, and dynamic installation and linking of best-in-class libraries You can install and inspect the cod in about a minute:

```
npm install hi_score
cd node_modules/hi_score
npm install
npm run prep-libs
npm test
npm run cover
google-chrome coverage/lcov-report/index.html
google-chome index.html
```

hi_score continues to evolve, and there are a few bits that still need to be hooked up. For example, the compression routines need a little more polish. However, we think it is already quite valuable and usable. And it's all libraries.

All xhi code uses typecasting to minimize type errors. Check out the code coverage on coveralls, and watch this blog for future hi_score announcements.

A parting thought

I recently saw this article on Angular which echoes many of the issues discussed above.

Cheers, Mike ■

A mushroom of many names

You may know them as portobello, button, baby bella and cremini, but believe it or not, these ubiquitous mushrooms all belong to the same species: *agaricus bisporus*.

When immature, it is a white, cherry-sized specimen known as champignon or button mushrooms. As it matures, it turns brown and is sold as cremini or baby bella mushrooms. Large, dark-colored specimens are known as portobello mushrooms and are frequently grilled to make vegan burgers.

* FOOD BIT is where we, enthusiasts of all edibles, sneak in a fun fact about food.