

Rust out your C

Carol (Nichols || Goulding)
@carols10cents



photo by flickr user celinet, cropped <https://flic.kr/p/8K8m9>

This is an experience report about what happened when I rewrote a C library in Rust. It will not be an intro to Rust, but I will touch on some Rust-specific concepts and explain them as we go.

Agenda

- 1 Caveats
- 2 Background
- 3 Techniques
- 4 Tricky Stuff
- 5 Benefits?



DO NOT

3

My first caveat: if you're thinking about rewriting a C library in Rust, do not. This is my cat disapproving of you. A rewrite is not something to take lightly, especially if the thing you want to rewrite is doing the job it needs to do just fine!

Bad reasons
to rewrite
your C in
Rust

- Cool kids
- I feel like it
- I'm bored
- Job security
- Carol said

4

Please don't leave this talk thinking "oh, Rust is the new hotness, I should tell my boss we have to stop working on everything to go off for a few months and rewrite everything in Rust!" No!!! Also, if you're the only person on your team interested in learning Rust, it may get you some job security, but it's not the most responsible thing to do in that situation.

Good reasons
to rewrite
your C in
Rust

- Performance
- Safety
- Lower maintenance costs
- Expand # of maintainers
- For fun, not work

5

If you have some code in C or another language, and need to change it, or it's slow, or it crashes a lot, or no one understands it anymore, THEN maybe a rewrite in Rust would be a good fit. I would also posit that more people are **able** to write production Rust than production C, so if your team **is** willing to learn Rust, it might actually expand the number of maintainers.

And then there's how I'm getting away with it— if you're doing a rewrite of something for fun, that's totally fine too.

I FIGHT FOR
THE USERS

Basically, keep the users of your software in mind when you're doing a rewrite. What will be the benefits for them?

Any time you
rewrite,
code will
get better.

7

Another caveat— Rust is not magic. Some of the effects I'm going to show you would have happened if I had chosen a different language to rewrite this library in, because rewriting gives you the benefit of hindsight.

Things I
knew before

- Rust
- Legacy code
- Testing

8

A confessional caveat— when I proposed this talk, these were the relevant things I knew...

Things I
DID NOT
know before

- C
- FFI
- zopfli

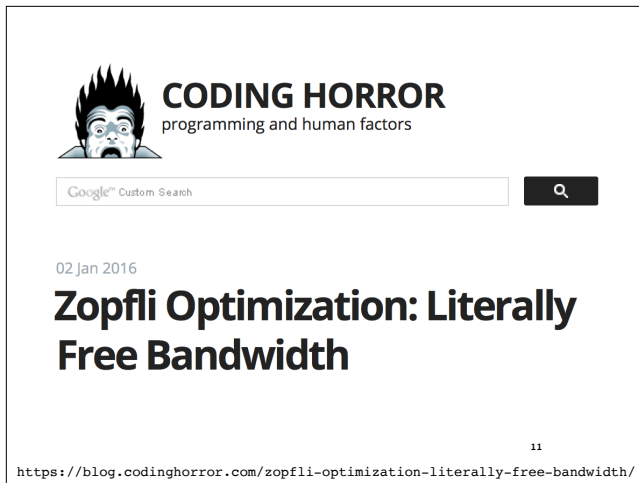
,

And these were the things I did NOT know. I haven't written C since college, I've been pretty terrified by it actually. I knew there was a foreign function interface between C and Rust but I hadn't tried it out. And the library I chose— I was not at all familiar with how it worked! But I learned and did everything I'm going to go over, which means YOU CAN TOO!

Background:
zopfli

10

Which brings us to the background info! Zopfli is a compression tool, like gzip or zlib. It was written by people at Google, and from what I can tell it's actually pretty good C code. I didn't have any trouble building it; there aren't any dependencies to set up. It's about 5K lines of C.

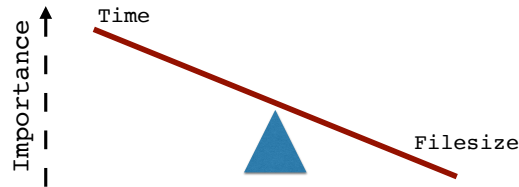


The reason you might want to use zopfli is that it creates smaller compressed files than gzip or zlib, but still follows the DEFLATE protocol,



which is what browsers know how to deflate. So using zopfli on your css, js, etc once can save bandwidth on every time those assets are transferred.

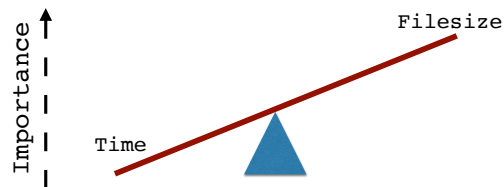
gzip



13


You might NOT want to use zopfli because it does take longer. Compared to gzip, which makes the tradeoff of taking less time to compress but potentially producing files that aren't as small as they could be,

zopfli



14

zopfli makes the other tradeoff, with the use case being you'll take the extra time to compress your files once and then save the extra file size every time your files are transferred.

 eliotsykes commented on Dec 8, 2015 +👍

The above benchmark is with Zopfli using its default 15 iterations. Results with a single Zopfli iteration bring the difference down to zlib being ~25 times faster than Zopfli. A single Zopfli iteration is almost as good as 15 Zopfli iterations in terms of the sample file size reductions.

Calculating			
zlib	9.000	1/100ms	
zopfli (1 iteration)	1.000	1/100ms	
<hr/>			
zlib	96.592	(± 6.2%) 1/s	486.000
zopfli (1 iteration)	4.195	(± 0.0%) 1/s	21.000

15

<https://github.com/rails/sprockets/pull/193>

I found out about zopfli in the context of Rails' asset pipeline— gzipping was actually disabled as part of the asset compilation process, and in the discussion around re-enabling it, zopfli was considered. If the filesize of all the Rails assets in all the world was smaller, that would save the internet a lot of bandwidth!

 schneems commented on Dec 8, 2015 Ruby on Rails member +👍

That's better, still a bit too slow to make the default I think. Maybe we can add it in and make it configurable.

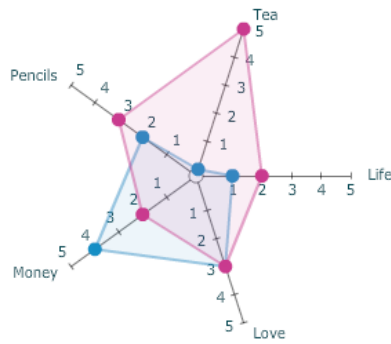
It looks like very little of the code is actually written in C. We could probably get a larger speed up by re-writing more of it in C and doing some benchmarking.

I have another concern with adding this in. I know libraries like Rails ship with gems with C extensions (nokogiri) and somehow they manage to play nice with other rubies like JRuby, but I'm not sure how exactly. We need to make sure we don't break jruby compatibility. There's no way to conditionally add something to a gemspec based on Ruby implementation (that I'm aware of). I also want to be cautious about adding c-extensions to dependencies. They take much longer to install, and by declaring it in the gemspec it would be installed even if someone was not using it. Deploy timeouts from too many c-extensions are a thing.

16

<https://github.com/rails/sprockets/pull/193>

But is it too slow? Would Rails developers complain? Could it be faster? And is the zopfli ruby gem a reasonable dependency to ask people to install? Would it work with JRuby? So now the tradeoffs are looking less like a seesaw and more like...



example radar chart demoing MIT Licensed flex component from
<http://www.boost.co.nz/blog/2009/05/flex-radar-chart-component/>

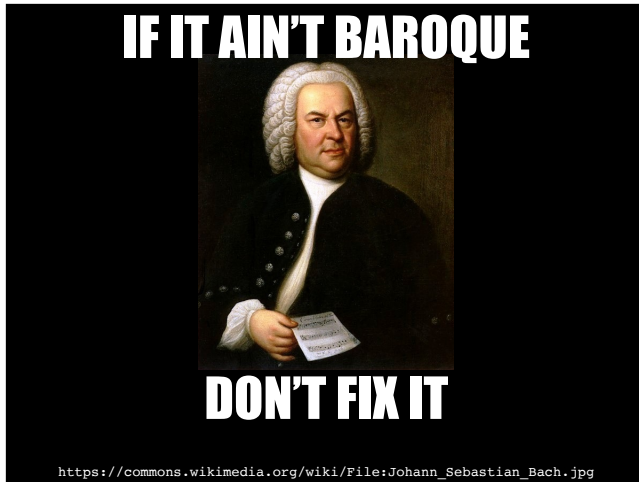
17

... comparing shapes in multiple dimensions. I thought Rust might be a good fit here, since it is fast and its cross-compilation story is good. I have no idea if it works better with JRuby though! And I had (and have) no idea if what I ended up with is a better choice than the C implementation of zopfli, but I figured this was as good a situation as any to try.

Techniques

18

The first technique I want to emphasize is a general legacy code technique—



you want to make sure that your rewrite isn't breaking anything and introducing new bugs!! But legacy code often doesn't come with automated tests, and zopfli did not either.

Golden Master Tests

20

So the first thing I did was make a golden master. In the case of zopfli, this meant I collected some files, compressed them and saved the results, and then with every change to the code, I could make sure I got the exact same compressed files. It's not as fine grained as unit tests, so it can be hard to figure out what the problem is when the files don't match, but it's better than not having any tests. An improvement I could have done but didn't would have been to do code coverage checks to make sure the files I picked were executing as much of the code as possible.

Makefile changes

```
+ZOPFLI_RUST_RELEASE := target/release/libzopfli.a  
ZOPFLILIB_SRC = src/zopfli/blocksplitter.c src/zopfli/cache.c...
```

21

So now into the actual code changes. I'm going to go pretty fast with this because I don't expect anyone to remember the changes exactly, but I wanted to show the scope and type of changes you'd need to make.

To the Makefile, I added a variable that held the Rust shared library we're going to create so that I could reference it similarly to the C source files.

Makefile changes

```
ZOPFLI_RUST_RELEASE := target/release/libzopfli.a  
ZOPFLILIB_SRC = src/zopfli/blocksplitter.c src/zopfli/cache.c...  
+.PHONY: target/release/libzopfli.a  
+target/release/libzopfli.a:  
+    cargo build --verbose --release
```

22

Then I told make how to build that library with cargo, using a PHONY target so that we always run cargo build when we make the C library.

Makefile changes

```
ZOPFLI_RUST_RELEASE := target/release/libzopfli.a
ZOPFLILIB_SRC = src/zopfli/blocksplitter.c src/zopfli/cache.c...
.PHONY: target/release/libzopfli.a
target/release/libzopfli.a:
    cargo build --verbose --release

-zopfli:
-   $(CC) $(ZOPFLILIB_SRC) $(ZOPFLIBIN_SRC) $(CFLAGS) -o zopfli
+zopfli: $(ZOPFLI_RUST_RELEASE)
+   $(CC) $(ZOPFLI_RUST_RELEASE) $(ZOPFLILIB_SRC) $(ZOPFLIBIN_SRC) $(CFLAGS) -o zopfli
```

23

Then for the main target, we made it depend on the shared library and link that in while building the C code.

Cargo.toml changes

```
+ [lib]
+ crate-type = ["staticlib"]
```

24

Cargo doesn't produce the .a file by default, but it will if we tell it we want a staticlib in the library's Cargo.toml.

Remove a function from C

```
size_t CalculateTreeSize(const unsigned* ll_lengths, const unsigned* d_lengths) {
    size_t result = 0;
    int i;

    for(i = 0; i < 8; i++) {
        size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                                         i & 1, i & 2, i & 4);
        if (result == 0 || size < result) result = size;
    }

    return result;
}
```

25

Now that we've got an empty Rust project being built and linked with C, let's say we want to move a function's implementation from C to Rust. This is a function from zopfli that I moved over. First, copy the whole C function.

Remove a function from C

```
extern size_t CalculateTreeSize(const unsigned* ll_lengths, const unsigned*
d_lengths);
```

26

Then remove the body, leaving only the signature. Add “extern” to the beginning and a semicolon at the end. This will let the remaining C code call this function.

Add a function to Rust

```
size_t CalculateTreeSize(const unsigned* ll_lengths, const unsigned* d_lengths) {
    size_t result = 0;
    int i;

    for(i = 0; i < 8; i++) {
        size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                                         i & 1, i & 2, i & 4);
        if (result == 0 || size < result) result = size;
    }
    return result;
}
```

27

Now paste the C code into a Rust code file.

Add a function to Rust

```
#[no_mangle]
#[allow(non_snake_case)]
size_t CalculateTreeSize(const unsigned* ll_lengths, const unsigned* d_lengths) {
    size_t result = 0;
    int i;

    for(i = 0; i < 8; i++) {
        size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                                         i & 1, i & 2, i & 4);
        if (result == 0 || size < result) result = size;
    }
    return result;
}
```

28

We have to tell Rust not to mangle the function name so that the C code can link to the symbol. Rust also isn't going to be happy about the non snake case name, but we're going to leave it this way for now, so turn off the warning about the naming convention for this function.

Add a function to Rust

```
#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(const unsigned* ll_lengths, const unsigned*
d_lengths) -> size_t {
    size_t result = 0;
    int i;

    for(i = 0; i < 8; i++) {
        size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                                         i & 1, i & 2, i & 4);
        if (result == 0 || size < result) result = size;
    }

    return result;
}
```

29

We'll need a pub extern fn at the beginning of the signature, and we'll move the return type that was there to the end after an arrow.

Add a function to Rust

```
#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: const unsigned*, d_lengths: const
unsigned*) -> size_t {
    size_t result = 0;
    int i;

    for(i = 0; i < 8; i++) {
        size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                                         i & 1, i & 2, i & 4);
        if (result == 0 || size < result) result = size;
    }

    return result;
}
```

30

At this point you could start trying to compile it and just let the Rust compiler tell you what needs fixed! Here's what it would say:
Argument names go before the types in Rust, so switch those around.

Add a function to Rust

```
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
    size_t result = 0;
    int i;

    for(i = 0; i < 8; i++) {
        size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
            i & 1, i & 2, i & 4);
        if (result == 0 || size < result) result = size;
    }

    return result;
}
```

31

We need to change the types a little bit— luckily there’s this Rust library called `libc` that defines aliases to map C types to Rust types, but we still need to change “unsigned” to “`c_uint`”.

Add a function to Rust

```
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
    let mut result = 0;
    int i;

    for(i = 0; i < 8; i++) {
        let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
            i & 1, i & 2, i & 4);
        if (result == 0 || size < result) result = size;
    }

    return result;
}
```

32

Next change the variable declarations to use “`let`” and make ones that need to be mutable

Add a function to Rust

```
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
    let mut result = 0;

    for i in 0..8 {
        let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
            i & 1, i & 2, i & 4);
        if (result == 0 || size < result) result = size;
    }

    return result;
}
```

33

Change the for loop to the Rust way of doing for loops, with a range

Add a function to Rust

```
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
    let mut result = 0;

    for i in 0..8 {
        let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
            i & 1, i & 2, i & 4);
        if result == 0 || size < result {
            result = size;
        }
    }

    return result;
}
```

34

Remove parens from the if condition and add braces

Add a function to Rust

```
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
    let mut result = 0;

    for i in 0..8 {
        let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                                     i & 1, i & 2, i & 4);
        if result == 0 || size < result {
            result = size;
        }
    }

    result
}
```

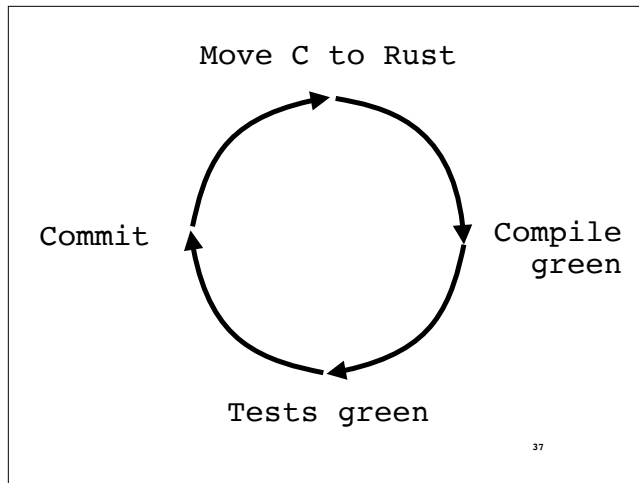
35

and in Rust we don't need to return if we're returning the last expression, we just need to not have a semicolon.

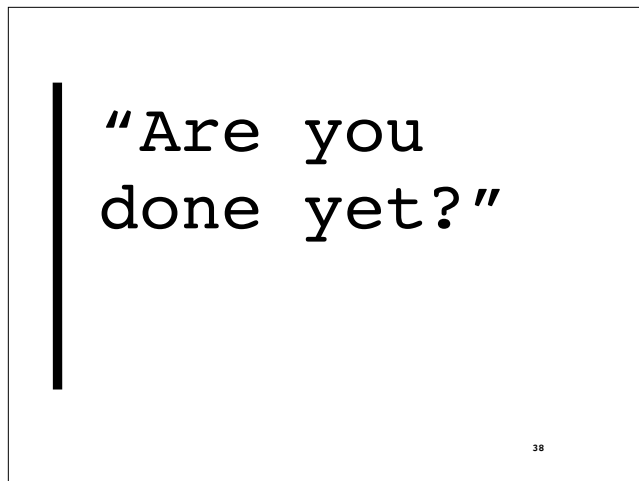
Incremental

36

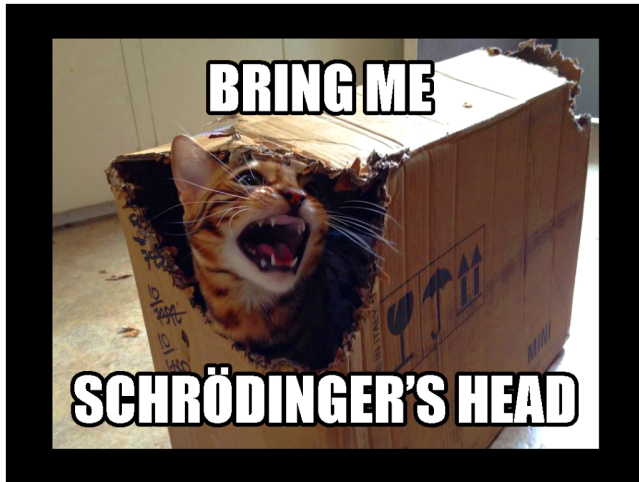
And then you repeat! Doing this incrementally, function by function, struct by struct is important. Not only do you only have to learn one little piece at a time, it's like a red-green-refactor TDD cycle,



except it's more like moving a function to rust, getting it to compile, getting the tests to pass, then committing. Taking lots of little baby steps and committing after each is great because when someone asks that ever-present question during rewrites,



as long as you only commit and push the code in a state where it's compiling and passing tests, you'll be in this sort of



superposition of done and not-done. You're done because you could ship what you have right now and switch to something else without having unrealized benefits from a sunk cost. But you're not done because the whole library isn't rewritten. So it gives you more flexibility in how you prioritize the rewrite if other more important tasks come up.

What if it
doesn't pass
the tests?

The other benefit of committing often after small changes is that, since you don't have fine-grained unit tests to tell you where the problem is, if your tests fail and you don't know why,

git checkout!
take a smaller
step.

41

it's not that big a deal to go back to a state where everything WAS working and try again. Probably try again with an even smaller step— many times when I broke everything, I was overconfident and was changing too much at once.

smaller

- Extract functions
- Make the C more like Rust first
- Don't make the Rust idiomatic at all, even when it seems easy

42

In this context, by smaller step I mean extracting functions from large functions so that you can move smaller pieces from C to rust, maybe making the C more like Rust first and getting that to pass the tests so that the actual change is more mechanical (ex: change complex C `for` loops to `while` loops more easily translated), and leave the Rust exactly like C even when it looks really simple to change pointers to slices— don't do it until after you have the tests passing!

✓ how
what

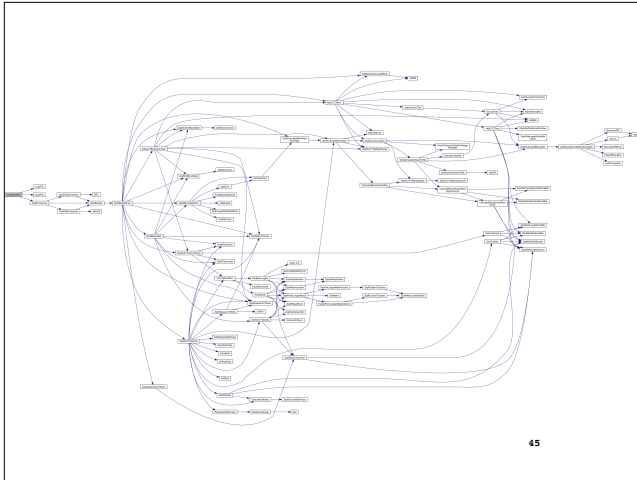
43

So now that we've covered how to change things from C to Rust, there are a few techniques I found to be useful for deciding what to change next.

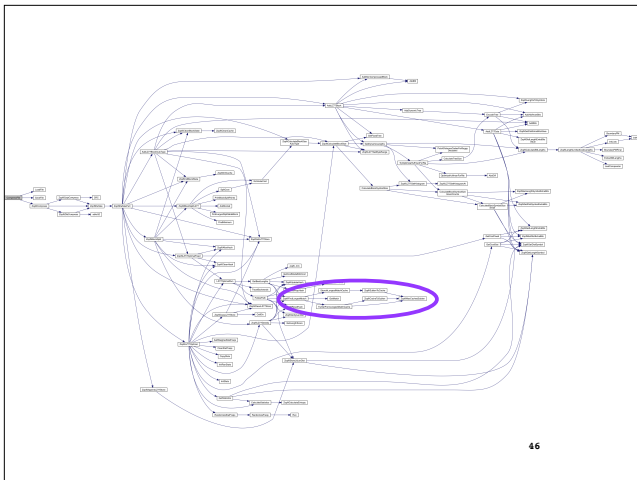
Follow the
call graph

44

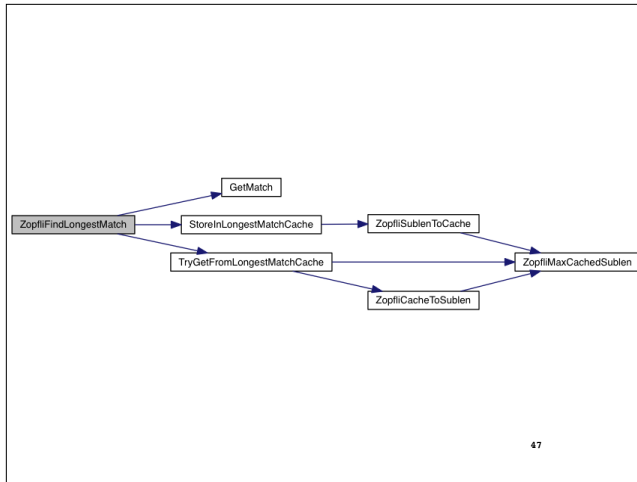
One is taking a look at the call graph of what functions call what other functions.



This is zopfli's full call graph, and from this you can see a few interesting things. You can see there's a main function that most of the work comes from, some interdependencies, and you can start to see "seams" – places where we might be able to carve off an independent chunk.



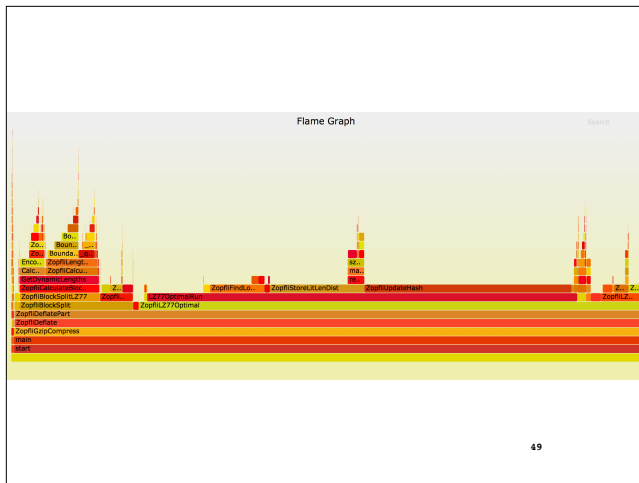
For instance, this part looks pretty isolated.



If we zoom in on that, we see this one function that calls the others, and they all call `ZopfliMaxCachedSublen`. These all have to do with caching a longest match, and there's a cache data structure associated with this. If we move all these data structures over to Rust, then this one entry point from C can stay the same but we can refactor and change the inner parts to be more idiomatically Rust without the C code being affected. This can get you more of the memory safety sooner than randomly choosing functions to rewrite.

Profile

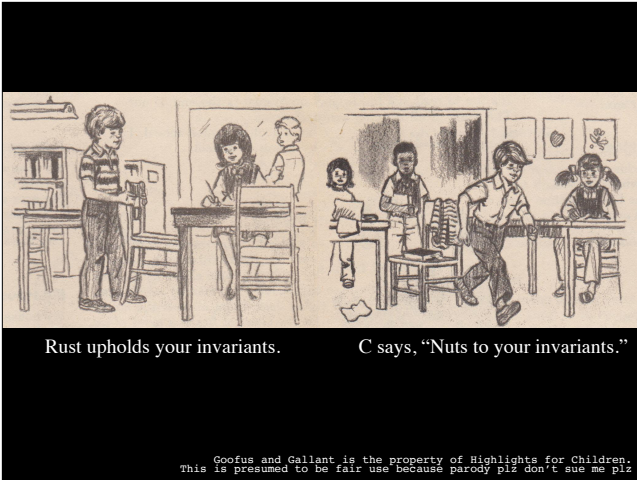
If your goal with the rewrite is to improve performance, then you might want to profile the code and work on the slowest parts first.



I didn't go about it this way, but if I had, I would have taken a look at this flame graph of the C code and probably looked into this `ZopfliUpdateHash` method first.

Tricky stuff

So now that we've gone over the straightforward "how"s and "what"s, let's take a look at a few of the problems I ran into along the way.



The most interesting problems were when the C code was doing something that the Rust compiler sees as likely violating the memory safety invariants it guarantees.

```
static void RandomizeFreqs(RanState*
state, size_t* freqs, int n) {

    int i;
    for (i = 0; i < n; i++) {

        if ((Ran(state) >> 4) % 3 == 0) {
            freqs[i] = freqs[Ran(state) % n];
        }

    }

}
```

52

For example, here's a function from the C code —

```
static void RandomizeFreqs(RanState*
state, size_t* freqs, int n) {
    int i;
    for (i = 0; i < n; i++) {
        if ((Ran(state) >> 4) % 3 == 0) {
            freqs[i] = freqs[Ran(state) % n];
        }
    }
}
```

53

The important part is this line which is writing to an element in an array while iterating through it and reading a different part of the array. And it's not even like, the previous item in the array, that we could easily see "oh that's safe"—it's a random element! We'd have to dig into what the `Ran` function is doing to know for sure.

```
for freq in freqs.iter_mut() {
    if (Ran(state) >> 4) % 3 == 0 {
        let index = (Ran(state) % n as
u32) as usize;
        *freq = freqs[index];
    }
}
```

54

So if we want to rewrite this as idiomatic, safe rust, we'd want to make freqs into a slice, then iterate mutably over that, but...

```

for freq in freqs.iter_mut() {
    if (Ran(state) >> 4) % 3 == 0 {
        let index = (Ran(state) % n as
u32) as usize;
        *freq = freqs[index];
    }
}

error: cannot use `freqs[..]` because it
was mutably borrowed [E0503]
    *freq = freqs[index];
      ^~~~~~
note: borrow of `*freqs` occurs here
for freq in freqs.iter_mut() {
    ^~~~~

```

55

Rust won't let us do borrow other items in the list immutably, since it's at the same time that we borrowed the list mutably, and we could be invalidating our iterator. So then you have a decision to make... do you keep the way the C code was doing it, which Rust considers unsafe, or do you rewrite the code in a safe way?

```

for i in 0..n {
    if (Ran(state) >> 4) % 3 == 0 {
        let index = (Ran(state) % n as
u32) as usize;
        freqs[i] = freqs[index];
    }
}

```

56

What I ended up doing was cheating to get around the borrow checker and iterating over the indexes instead of the items in the slice. This way, the mutable borrow is only valid for the innermost part, not the whole loop. This does lose some of the protections Rust provides; if n isn't actually the length of the array, the program could crash with an array index out of bounds error.

RandomizeFreqs...
shuffle?

57

Something better would be to actually take a look at what this code's goal is and see if we can accomplish this in a different way... if we're **just** randomizing the frequencies, could we call shuffle? This didn't work for me, the resulting compressed files changed, but idk if the changes matter. This is a tricky part of legacy code in general – when do you actually **want** to change behavior?

Type conversions 🤖

```
double GetBestLengths(..., float* costs, ...) {  
    double newCost = costmodel(in[i]) + costs[j];  
    if (newCost < costs[j + 1]) {  
        costs[j + 1] = newCost;  
    }  
    ...  
}
```

58

So another thing that became clear when I rewrote the C code in Rust is converting between types. C converts between types implicitly, which makes it harder to see that it's happening— it's not very clear in this code, but there are conversions between floats and doubles going on here.

Type conversions 🤖

```
double GetBestLengths(..., float* costs, ...) {
    double newCost = costmodel(in[i]) + costs[j];
    if (newCost < costs[j + 1]) {
        costs[j + 1] = newCost;
    }
    ...
}

fn GetBestLengths(..., costs: *mut c_float, ...) -> c_double {
    let newCost = costmodel(in[i]) + costs[j] as c_double;
    if newCost < costs[j + 1] as c_double {
        costs[j + 1] = newCost as c_float;
    }
    ...
}
```

59

Rust favors being explicit over implicit, which means the Rust code needs a lot more type conversions in the form of `as type`s.

Type conversions 🤖

```
double GetBestLengths(..., float* costs, ...) {
    double newCost = costmodel(in[i]) + costs[j];
    if (newCost < costs[j + 1]) {
        costs[j + 1] = newCost;
    }
    ...
}

fn GetBestLengths(..., costs: *mut c_float, ...) -> c_double {
    let newCost = costmodel(in[i]) + costs[j] as c_double;
    if newCost < costs[j + 1] as c_double {
        costs[j + 1] = newCost as c_float;
    }
    ...
}
```

60

The interesting part to me is where newCost, a double, is being converted to an element of costs, floats, but compared at a higher precision as a double, but then the newCost is stored with a lower precision as a float in costs! Is this a bug? If I change it so everything is doubles or floats, the golden master files change, so it does change behavior, but is it changing it in a way to be better or worse? I'm not sure! This is another case where your knowledge of how the library works has to increase in order to be confident with making a change.

0 means...

- 0
- 0.0
- False
- Null pointer
- End of string
- Sadness
- Frustration

61

Another thing I found super frustrating and tricky was the overloading, in C, of the meaning of 0 to be the integer 0, the floating point 0, false, null pointer, etc etc. Often the same variable with the value 0 was used in different ways, which made understanding and translating the code difficult for me! I don't know how C devs live like this.

Benefits?

62

So — did I see any benefits from the rewrite in Rust?

Safety

63

Fewer possibilities of null pointers— you could just forget to check once in C and oopsie! Most places, I was able to change this so that Rust could guarantee I would have something.

Clarity

64

C is not very object oriented; I was able to extract objects and methods on them to regroup functionality. Rust is able to return tuples; C passes mutable arguments, which I find confusing. Separating the different meanings of 0 into different concepts was awesome! These are kinds of things I think rewriting into most anything but C could get you, though.

Less code

65

C is pretty bare bones; Rust does a lot more for you. Even though Rust's stdlib can be described as “small”!

```
#define ZOPFLI_APPEND_DATA(value, data, size) {\
    if (!((*size) & ((*size) - 1))) {\
        /*double alloc size if it's a power of two*/\
        (*data) = (*size) == 0 ? malloc(sizeof(**data))\
                               : realloc((*data), (*size) * 2 *
sizeof(**data));\
    }\
    (*data)[(*size)] = (value);\
    (*size)++;\
}
```

66

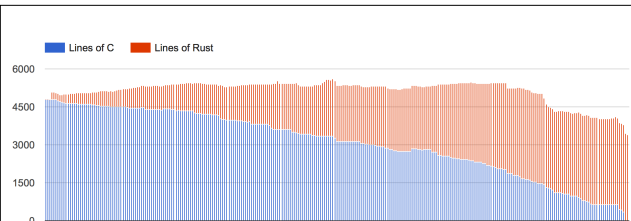
For example, every time zopfli wanted to add a value to the data output array, it would check to see if the size is a power of two, and if so, it would reallocate double that amount.

```
#define ZOPFLI_APPEND_DATA(value, data, size) {\
    if (!((*size) & ((*size) - 1))) {\
        /*double alloc size if it's a power of two*/\
        (*data) = (*size) == 0 ? malloc(sizeof(**data))\
                                : realloc((*data), (*size) * 2 *\
sizeof(**data));\
    }\
    (*data)[(*size)] = (value);\
    (*size)++;\
}
```

`data.push(value)`

67

In Rust, the `std::Vec` type does this under the hood so instead of having to maintain a function like `ZOPFLI_APPEND_DATA` and call it all the time, you can just push onto the vec!



4777 LOC in C ->
3399 LOC in Rust =
71%

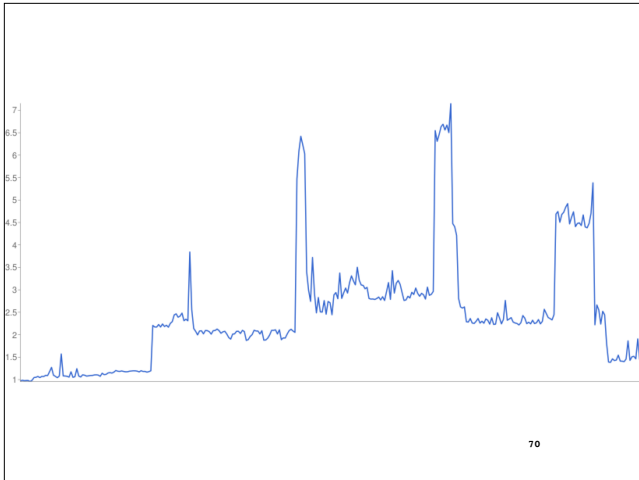
68

This is the graph over time of how much code there was in C and Rust as I worked; by the end when I got it all into Rust, it was 71% of the lines and could probably continue to go down if I were to make the Rust more idiomatic and recognize other things the `std::lib` does.

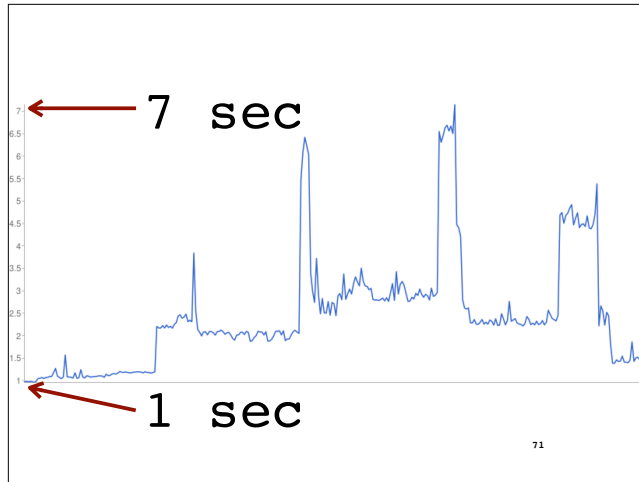
Performance

69

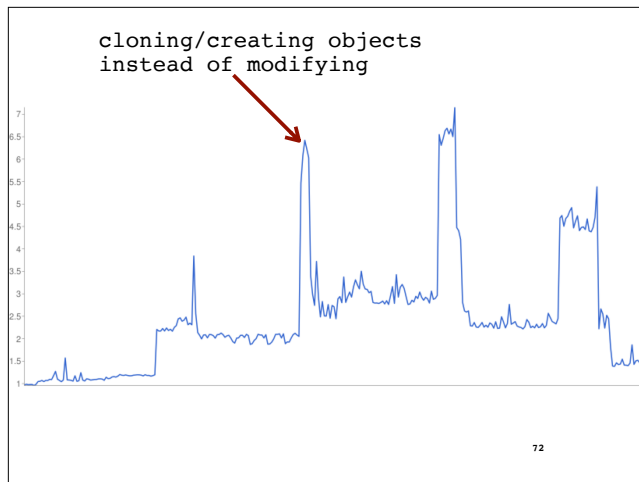
The big question: is Rust as fast as C? (Or, really, is the Rust Carol translated as fast as pretty good C?)



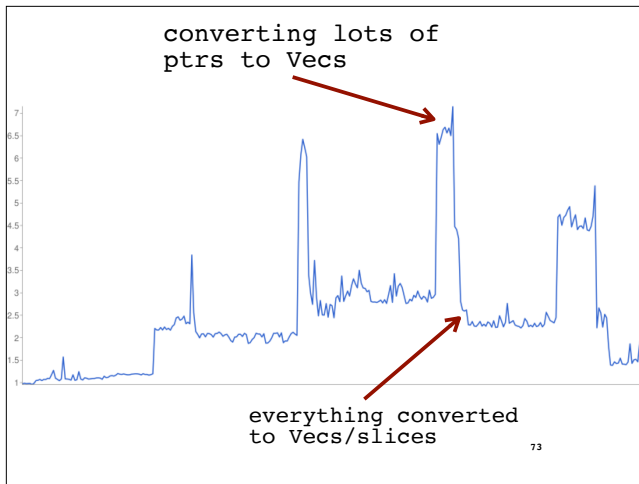
Well, I haven't gotten it back down to C speeds yet, but it's pretty close. This is the performance at each commit of mine.



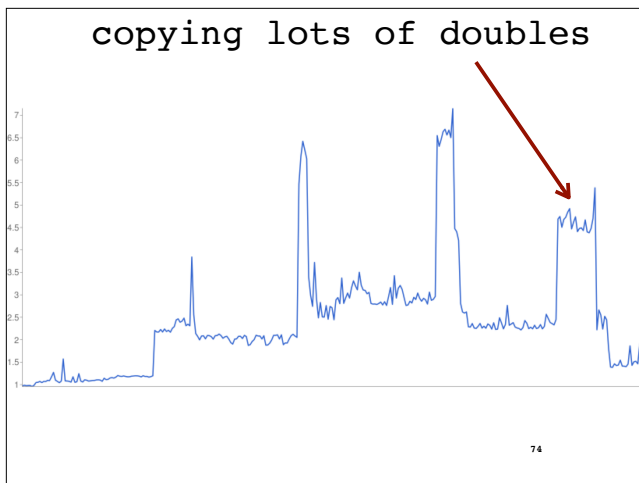
C code took about 1 sec on my golden master tests. This graph is rough, only ran the tests once per commit.



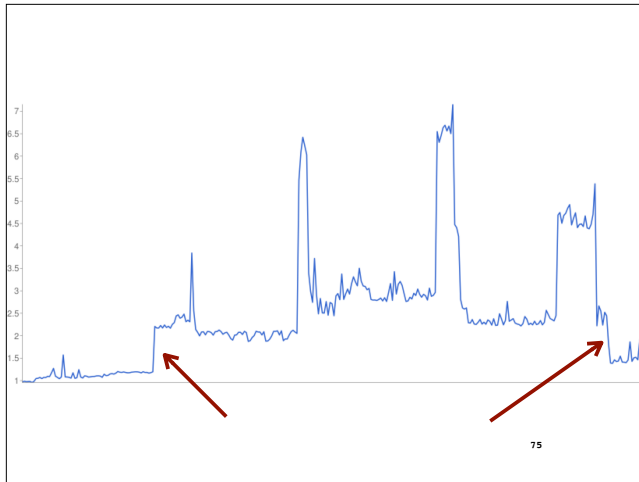
Digging into some of the spikes and why the time went up, this spike was because I was using more of a functional style and allocating new objects instead of modifying the ones I already had.



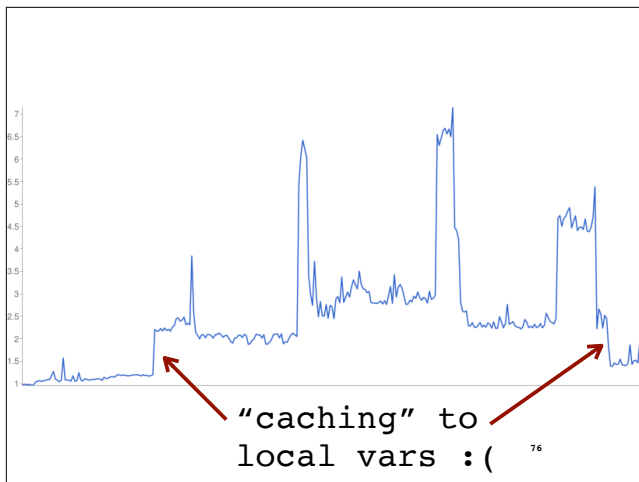
This spike was while I was in the middle of transitioning a data structure from pointers to vecs, so I was doing a lot of switching back and forth. Once I got all of the uses of that structure in Rust and wasn't converting between the two types, performance improved again.



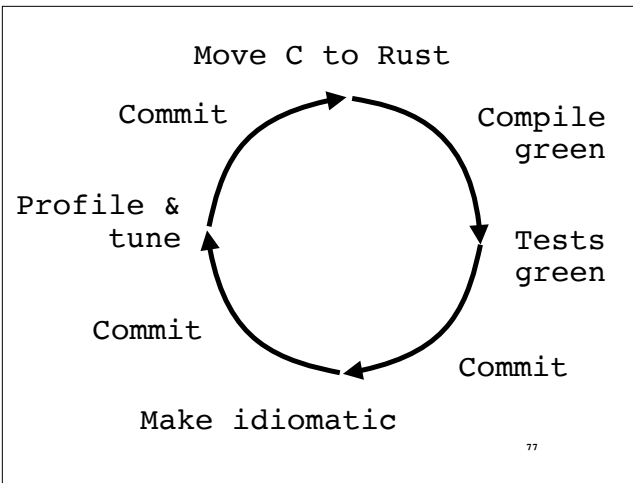
This spot was when I converted some functions to pass a Rust struct around instead of a raw pointer. That struct had a fixed-length array of a few hundred doubles (f64). By default, objects get moved into functions that call them, but primitive types get copied— and structs that are made up of primitives like arrays and f64s are also copied. So when I changed it to borrow that struct instead of copy, performance got better.



Saved the most interesting one for last— I didn't notice this jump in performance near the beginning. I wasn't paying very close attention and it wasn't a huge jump, but I looked into it at the end and it turns out I had introduced a bug—



I thought I was storing values to a cache, turns out I was only caching to local variables :(



So do what I say and not what I did— if your goal is to maintain or improve performance, your iterations should probably look more like this: after each function gets moved over, spend time checking performance and catching those regressions earlier instead of at the end like I did.

C-like Rust is
slower than
idiomatic Rust?

78

My best guess for the rest of the slowness is that the Rust code is still pretty C-like, and isn't taking advantage of optimizations that idiomatic Rust could be. I'd like to do more work to see if this is the case.

Future work

- Remove all ``unsafe``
- Use more iterators
- Stream input/output
- Refactor forever

79

Other things I'd like to do that I ran out of time to do before this talk are getting rid of all the uses of ``unsafe`` so that we're getting all the safety Rust could provide, using more iterators over elements instead of for loops to not have to worry about going past the end of arrays, Streaming is something the C library could use that I think Rust would make easier, so that's an enhancement actually, and of course I could move code around forever, as we all could on all of our code.

So... should you rewrite your C library in Rust?

- _ (ツ) _ / -

80

I dunno, maybe!

my point:

- Incremental rewrites from C to Rust are possible.
- Have reasons for rewriting and measure progress against the reasons.

81

What I hope you take away from my less-than-convincing results is that it is possible to rewrite C libraries in Rust in an incremental manner, and that if you choose to undertake a rewrite, you should have a good reasons for doing so and you should measure against those goals along the way.

References (is.gd/c_rust)

- [Repo of my code](#)
- These slides
- [FFI chapter in The Rust Programming Language book](#)
- [Rust FFI Omnibus](#)
- [Working Effectively with Legacy Code](#) by Michael Feathers

82

Thank
You

Carol (Nichols || Goulding)
@carols10cents

