A crash course in just-in-time (JIT) compilers

By Lin Clark

Posted on February 28, 2017 in A cartoon intro to WebAssembly, JavaScript, Performance, and WebAssembly

This is the second part in a series on WebAssembly and what makes it fast. If you haven't read the others, we recommend starting from the beginning.

JavaScript started out slow, but then got faster thanks to something called the JIT. But how does the JIT work?

How JavaScript is run in the browser

When you as a developer add JavaScript to the page, you have a goal and a problem.

Goal: you want to tell the computer what to do.

Problem: you and the computer speak different languages.

You speak a human language, and the computer speaks a machine language. Even if you don't think about JavaScript or other high-level programming languages as human languages, they really are. They've been designed for human cognition, not for machine cognition.

So the job of the JavaScript engine is to take your human language and turn it into something the machine understands.

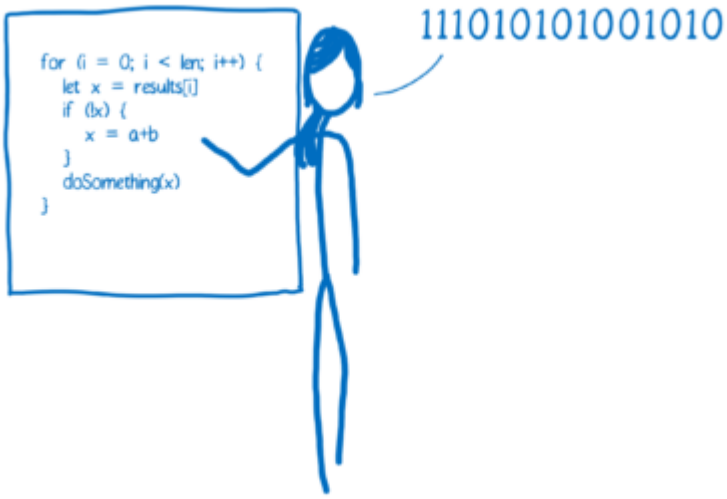
I think of this like the movie Arrival, where you have humans and aliens who are trying to talk to each other.

In that movie, the humans and aliens don't just do word-for-word translations. The two groups have different ways of thinking about the world. And that's true of humans and machines too (I'll explain this more in the next post).

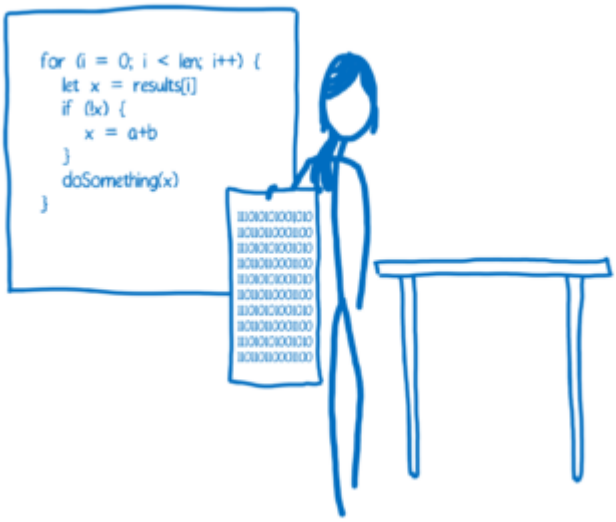
So how does the translation happen?

In programming, there are generally two ways of translating to machine language. You can use an interpreter or a compiler.

With an interpreter, this translation happens pretty much line-by-line, on the fly.



A compiler on the other hand doesn't translate on the fly. It works ahead of time to create that translation and write it down.



There are pros and cons to each of these ways of handling the translation.

Interpreter pros and cons

Interpreters are quick to get up and running. You don't have to go through that whole compilation step before you can start running your code. You just start translating that first line and running it.

Because of this, an interpreter seems like a natural fit for something like JavaScript. It's important for a web developer to be able to get going and run their code quickly.

And that's why browsers used JavaScript interpreters in the beginning.

But the con of using an interpreter comes when you're running the same code more than once. For example, if you're in a loop. Then you have to do the same translation over and over and over again.

Compiler pros and cons

The compiler has the opposite trade-offs.

It takes a little bit more time to start up because it has to go through that compilation step at the beginning. But then code in loops runs faster, because it doesn't need to repeat the translation for each pass through that loop.

Another difference is that the compiler has more time to look at the code and make edits to it so that it will run faster. These edits are called optimizations.

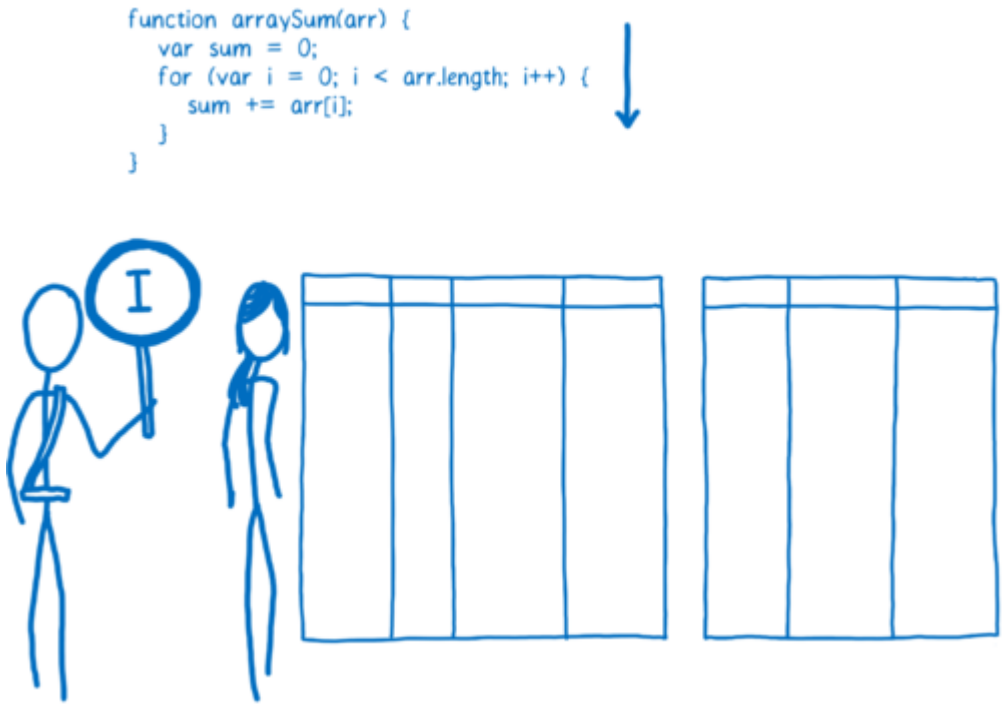
The interpreter is doing its work during runtime, so it can't take much time during the translation phase to figure out these optimizations.

Just-in-time compilers: the best of both worlds

As a way of getting rid of the interpreter's inefficiency—where the interpreter has to keep retranslating the code every time they go through the loop—browsers started mixing compilers in.

Different browsers do this in slightly different ways, but the basic idea is the same. They added a new part to the JavaScript engine, called a monitor (aka a profiler). That monitor watches the code as it runs, and makes a note of how many times it is run and what types are used.

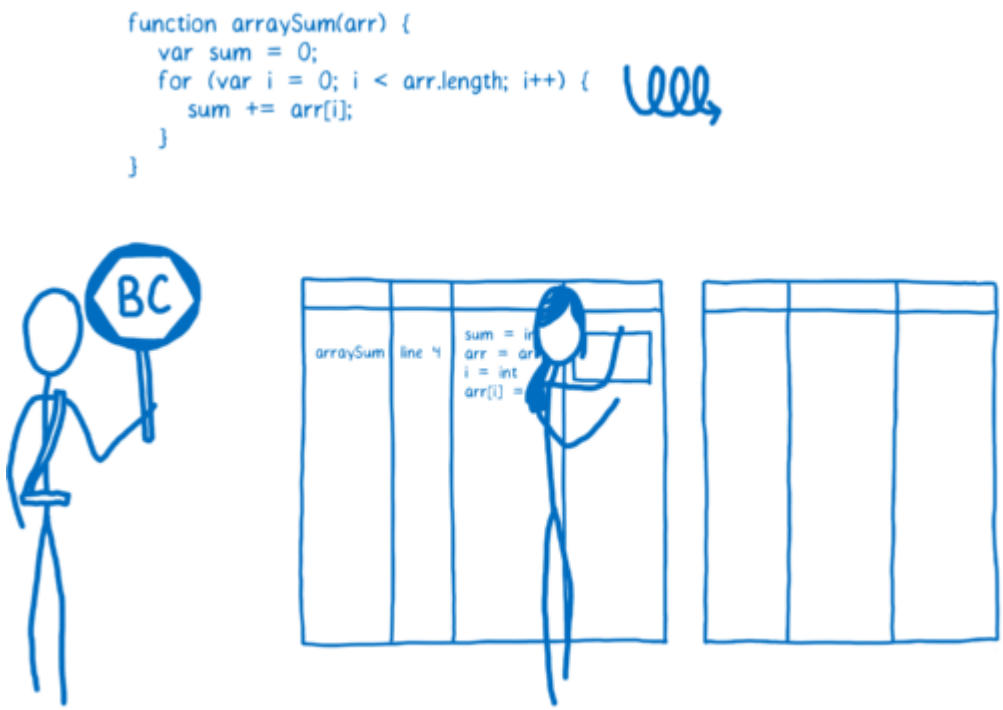
At first, the monitor just runs everything through the interpreter.



If the same lines of code are run a few times, that segment of code is called warm. If it's run a lot, then it's called hot.

Baseline compiler

When a function starts getting warm, the JIT will send it off to be compiled. Then it will store that compilation.



Each line of the function is compiled to a “stub”. The stubs are indexed by line number and variable type (I’ll explain why that’s important later). If the monitor sees that execution is hitting the same code again with the same variable types, it will just pull out its compiled version.

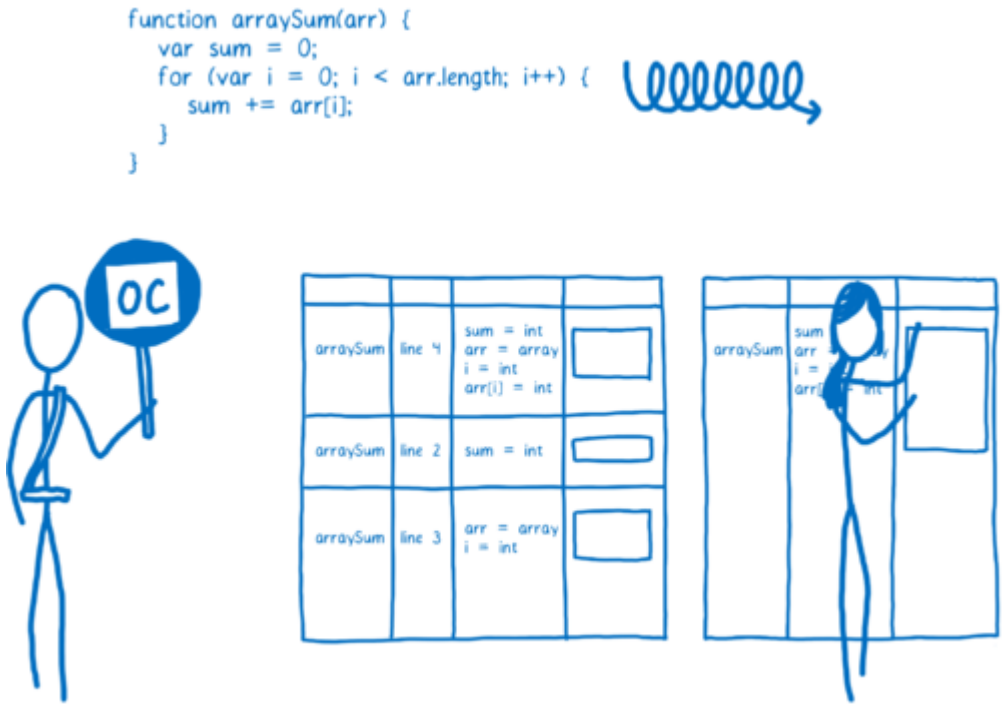
That helps speed things up. But like I said, there’s more a compiler can do. It can take some time to figure out the most efficient way to do things… to make optimizations.

The baseline compiler will make some of these optimizations (I give an example of one below). It doesn’t want to take too much time, though, because it doesn’t want to hold up execution too long.

However, if the code is really hot—if it’s being run a whole bunch of times—then it’s worth taking the extra time to make more optimizations.

Optimizing compiler

When a part of the code is very hot, the monitor will send it off to the optimizing compiler. This will create another, even faster, version of the function that will also be stored.



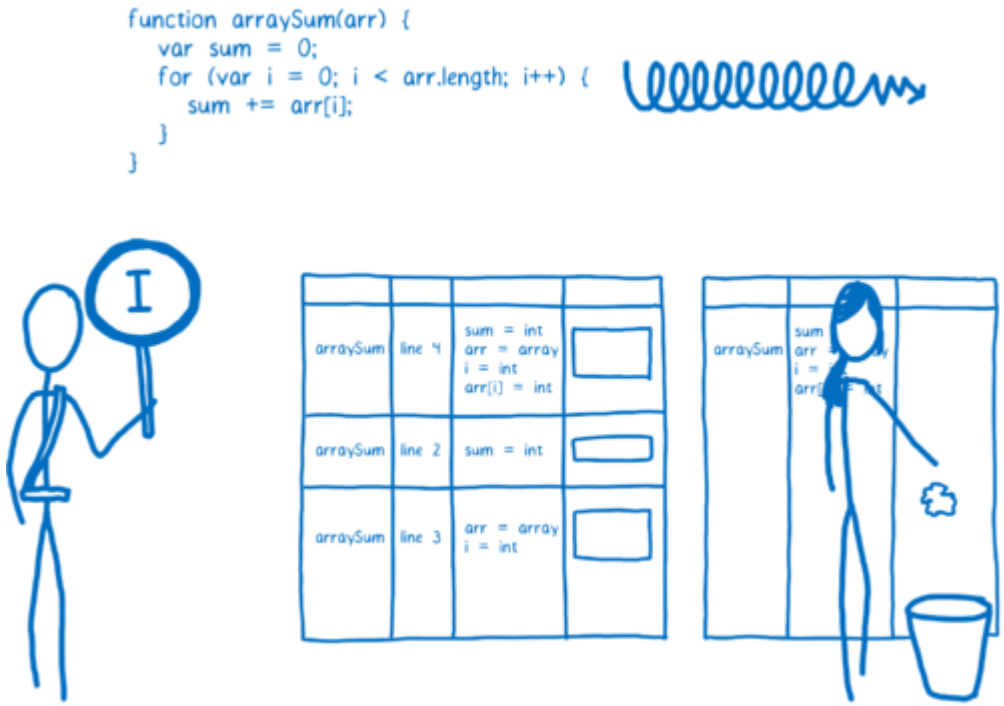
In order to make a faster version of the code, the optimizing compiler has to make some assumptions.

For example, if it can assume that all objects created by a particular constructor have the same shape —that is, that they always have the same property names, and that those properties were added in the same order— then it can cut some corners based on that.

The optimizing compiler uses the information the monitor has gathered by watching code execution to make these judgments. If something has been true for all previous passes through a loop, it assumes it will continue to be true.

But of course with JavaScript, there are never any guarantees. You could have 99 objects that all have the same shape, but then the 100th might be missing a property.

So the compiled code needs to check before it runs to see whether the assumptions are valid. If they are, then the compiled code runs. But if not, the JIT assumes that it made the wrong assumptions and trashes the optimized code.



Then execution goes back to the interpreter or baseline compiled version. This process is called deoptimization (or bailing out).

Usually optimizing compilers make code faster, but sometimes they can cause unexpected performance problems. If you have code that keeps getting optimized and then deoptimized, it ends up being slower than just executing the baseline compiled version.

Most browsers have added limits to break out of these optimization/deoptimization cycles when they happen. If the JIT has made more than, say, 10 attempts at optimizing and keeps having to throw it out, it will just stop trying.

An example optimization: Type specialization

There are a lot of different kinds of optimizations, but I want to take a look at one type so you can get a feel for how optimization happens. One of the biggest wins in optimizing compilers comes from something called type specialization.

The dynamic type system that JavaScript uses requires a little bit of extra work at runtime. For example, consider this code:

```
function arraySum(arr) {
  var sum = 0;
  for (var i = 0; i < arr.length; i++) {
    sum += arr[i];
  }
}
```

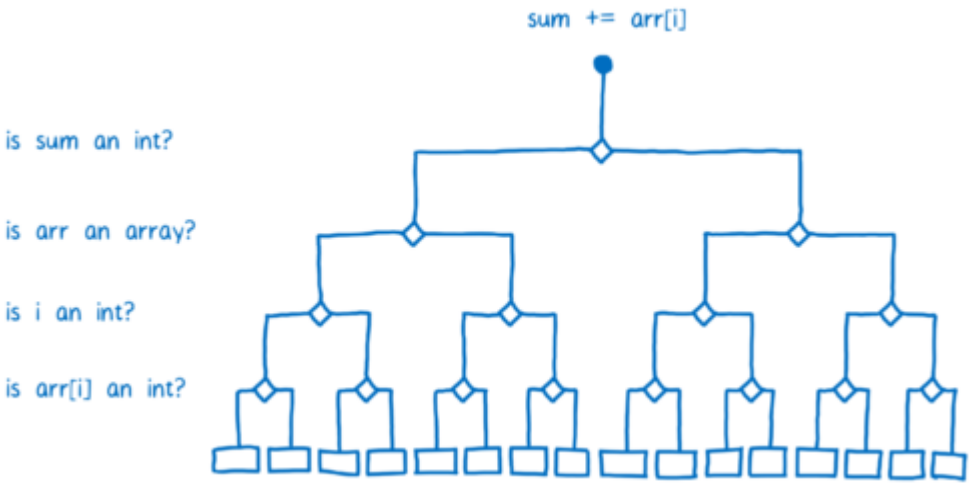
The += step in the loop may seem simple. It may seem like you can compute this in one step, but because of dynamic typing, it takes more steps than you would expect.

Let's assume that `arr` is an array of 100 integers. Once the code warms up, the baseline compiler will create a stub for each operation in the function. So there will be a stub for `sum += arr[i]`, which will handle the += operation as integer addition.

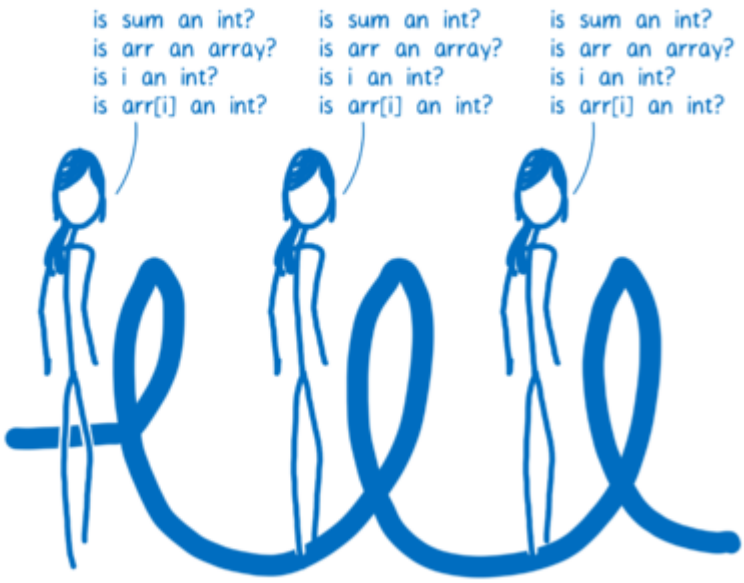
However,`sum` and `arr[i]` aren't guaranteed to be integers. Because types are dynamic in JavaScript, there's a chance that in a later iteration of the loop, `arr[i]` will be a string. Integer addition and string concatenation are two very different operations, so they would compile to very different machine code.

The way the JIT handles this is by compiling multiple baseline stubs. If a piece of code is monomorphic (that is, always called with the same types) it will get one stub. If it is polymorphic (called with different types from one pass through the code to another), then it will get a stub for each combination of types that has come through that operation.

This means that the JIT has to ask a lot of questions before it chooses a stub.

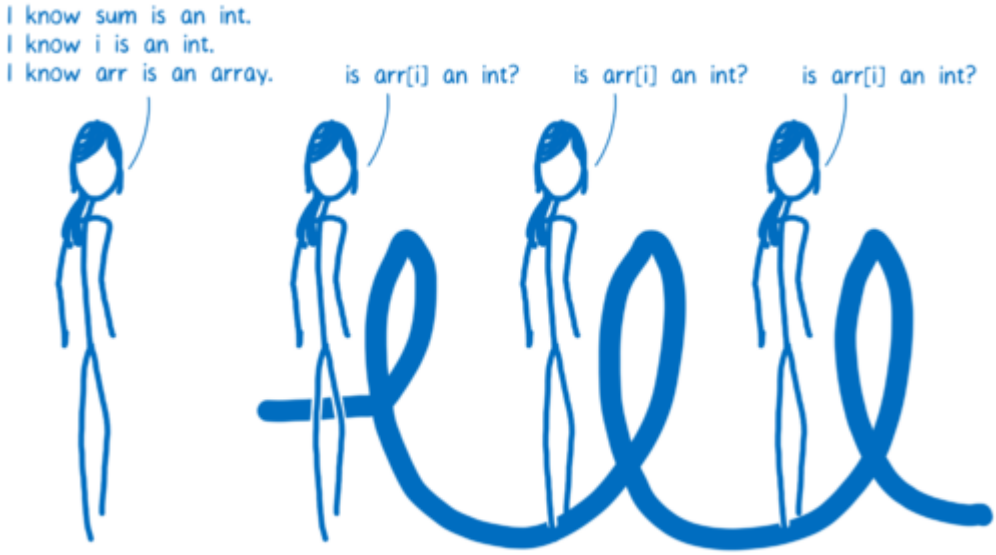


Because each line of code has its own set of stubs in the baseline compiler, the JIT needs to keep checking the types each time the line of code is executed. So for each iteration through the loop, it will have to ask the same questions.



The code would execute a lot faster if the JIT didn't need to repeat those checks. And that's one of the things the optimizing compiler does.

In the optimizing compiler, the whole function is compiled together. The type checks are moved so that they happen before the loop.



Some JITs optimize this even further. For example, in Firefox there's a special classification for arrays that only contain integers. If `arr` is one of these arrays, then the JIT doesn't need to check if `arr[i]` is an integer. This means that the JIT can do all of the type checks before it enters the loop.

Conclusion

That is the JIT in a nutshell. It makes JavaScript run faster by monitoring the code as it's running it and sending hot code paths to be optimized. This has resulted in many-fold performance improvements for most JavaScript applications.

Even with these improvements, though, the performance of JavaScript can be unpredictable. And to make things faster, the JIT has added some overhead during runtime, including:

- optimization and deoptimization
- memory used for the monitor's bookkeeping and recovery information for when bailouts happen
- memory used to store baseline and optimized versions of a function

There's room for improvement here: that overhead could be removed, making performance more predictable. And that's one of the things that WebAssembly does.

In the next article, I'll explain more about assembly and how compilers work with it.

About Lin Clark

Lin works in Advanced Development at Mozilla, with a focus on Rust and WebAssembly.

https://twitter.com/linclark
@linclark

More articles by Lin Clark...

14 comments

Erik H Reppen

Thank you for this. I haven't even finished reading yet but I've been writing JS for over a decade and I'm looking forward to getting a good mix of plain-English + non-handholding on some of the details of JIT compilers.

February 28th, 2017 at 20:13