js.rs – A Rustic JavaScript Interpreter

Terry Sun, Sam Rossi

April 25, 2016

Introduction

JavaScript is an incredibly widespread language, running on virtually every modern computer and browser, and interpreters such as NodeJS allow JavaScript to be used as a server-side language. Unfortunately, modern implementations of JavaScript engines are typically written in C/C++, languages reliant on manual memory management. This results in countless memory leaks, bugs, and security vulnerabilities related to memory mismanagement.

Weve chosen to build a prototype server-side JavaScript interpreter in Rust, a new systems programming language for building programs with strong memory safety guarantees and speeds comparable to C++. Our interpreter runs code either from source files or an interactive REPL (read-evaluate-print-loop), similar to the functionality of existing server-side JavaScript interpreters. We intend to demonstrate the viability of using Rust to implement JavaScript by implementing a core subset of language features. To that end, weve tested our coverage using Googles Sputnik test suite, an ECMAScript 5 conformance test suite.

The Rust Language

According to the official website, Rust is "systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety"[1]. Designed to be safer than traditional systems programming languages, Rust's creator described Rust as designed for "frustrated C++ developers"[2]. To that end, Rust guarantees memory safety in a relatively unique way compared to most mainstream programming languages; while traditional memory-safe languages use garbage collection, which incurs significant runtime costs, Rust using a combination of compile-time techniques including a nuanced type system, tracking of pointer lifetimes, and several other cutting-edge techniques developed by by programming languages researchers over the years.

Approach

Parser

The first part of the interpreter that we implemented was the parser, which takes in a string of JavaScript and generates an "Abstract Syntax Tree", or AST. An AST contains the structure and the content of a program, but not the specific syntactic components such as whitespacing.

Traditionally, the canonical way of writing a parser is to use a "parser generator". Implementing a parser with a parser generator consists of defining the grammar of the language (i.e. the valid tokens of the language and what sequences of tokens are valid); the parser

generator then generate code to parse the grammar, which is then used as a library. Although parser generators tend to be a bit slower in practice than writing an equivalent parser manually, using a parser generate greatly increases the rate of development due to the ease of use. Because of the scope and time limitations of our project, we opted to use a parser generator rather than writing a custom parser.
The two most widely used parser generators are YACC and Bison, which are implemented in C. Neither of these would be suitable for our project, as we intended to use only Rust libraries in our interpreter to maximize the safety guarantees. After some research, we decided to use LALRPOP, a pure Rust LR(1) parser generator[3].

Garbage Collector

Like other interpreted languages, JavaScript relies on garbage collection to manage the memory used by a running program. Although it would have been possible for us to write our own garbage collector, a master's student with the same advisor as us was working on implementing a JavaScript garbage collection library in Rust for his thesis. Integrating with the garbage collection library seemed mutually beneficial, so we opted not to write our own garbage collector in favor of using French Press.

Runtime

In terms of time and amount of code, the most signficant part of our work was the actual evaluation of the JavaScript code, which is generally referred to as the "runtime system", or juse the "runtime". In order to effectively follow the proper semantics of JavaScript code, we made heavy use of the Mozilla Developer Network's official JavaScript documentation[4]. This greatly reduced the amount of effort we had to spend on determining things like the proper precendence of operators and the correct implementation of implicit type coercions.

Functions, Scoping, and Closures

One of the most difficult parts of implementing the runtime was ensuring that it exhibited the correct behavior with regards to functions and scoping. For simple cases, French Press handled the correct scoping for local variables. However, in the case of closures, the standard scoping rules would not suffice. The following code sample demonstrates such a case:

```
function f() {
    var x = 0;
    return function() {
        return x++;
    };
}

var g = f();
console.log(g());
console.log(g());
```

Normally, when a function returns, all of its local variables are no longer in scope, so the garbage collector can deallocate the memory associated them. However, in the above example, when `f` is called, it returns a new function which has access to `x`. This means that `x` cannot be garbage collected at least until after `g` is no longer in scope. In order to correctly execute code cases like this, we had to add functionality for our code to detect when a function is a closure and inform the garbage collector of the special status of that function's scope.

Results

Although we did not have time to implement 100% of the JavaScript language, we implemented a significant subset of features which provide more than enough to write interesting and useful programs.

Language Features

We successfully implemented parsing and evaluation of the following types of expressions and statements:

Literal expressions

- Boolean: `true`, `false`
- Numeric: e.g. `-4`, `7.17`, `NaN`
- String: e.g. `"abcĭ234"`, `'foo bar baz'`
- Object: e.g. `{ x: 3, y: { z: "hello" } }`
- Array: e.g. `[1, "hello", x]`
- `null`
- `undefined`

Operator expressions

- Arithmetic: `+`, `-`, `*`, `/`, `%`
- Incrementation: `++`, `--`
- Boolean logical: `&&`, `||`, `!`
- Bitwise logical: `&`, `|`, , `^`
- Shifts: `>>>`, `>>`, `<<`
- Inequalties: `>`, `>=`, `<`, `<=`
- Equalities: `==`, `!=`, `===`, `!==`
- Assignment: `=`, `+=`, `&&=`, ...
- `instanceof`
- `typeof`

Function-related expressions

- Named function definition: e.g. `function f(x) { return x + 1; }`
- Anonymous function definition: e.g. `function(x, y) { return x + y; }`
- Function calls: e.g. `foo(a, b)`

Object-related expressions

- Instance variable access: e.g. `foo.bar`
- Key-indexed access: e.g. `foo[bar]`
- Constructors: e.g. `new foobar(x, y)`

Mutation statements

- Assignment: e.g. `x = y`, `foo[bar] = 23`
- Declarations: e.g. `var x = z`

Control-flow statements

- `if`/`else if`/`else`
- `while`
- `for`
- `break`
- `continue`
- `return`
- `try`/`catch`/`finally`
- `throw`

Standard library

We implemented a small standard library for our interpreter based on some of the most widely-used features the official JavaScript standard library

Printing

Official JavaScript interpreters do console-based I/O through the global `console` object, which has methods such as `log()` and `error`. Because implementing the entire native `console` object would have taken significant time and distracted us from implementing more widely-used features, we implemented a global `log()` function which behaves like `console.log()`.

Prototypes

JavaScript typically provides a number of built-in prototypes, including `String`, `Array`, `Object`, and `Function`. Because the majority of the other features could be easily implemented natively, we decided to build in a simple `Array` prototype to demonstrate our implementation of prototype functionality in our interpreter.

Coverage

To test how complete our coverage of the JavaScript standard was, we built a framework to run the Google Sputnik test suite[5] on our interpreter. The test suite covers the ECMAScript 3 subset of the ECMAScript 5 standard. We used two different metrics to analyze the coverage of our interpreter.

Category-based coverage

Sputnik defines several categories of tests, each with various depths of subcategories (for example, the "Expressions" category contains, among others, a "Postfix Expressions" subcategory, which in turn contains the subcategories "Postfix Increment Operator" and "Postfix Decrement Operator"). Overall, there are 111 leaf categories (i.e. categories which do not contain other categories).

In order to determine what percentage of the JavaScript language features we had implemented, we counted the number of leaf categories in which we had passed some of the tests. Of the 111, we found that we had coverage in 73 of the categories, or 65.8% of them. This indicates that we while we had not implemented a full-fledged JavaScript interpreter, we covered a sizable portion of the languages features.

Raw coverage

Ethical and Privacy Considerations

Although our project is novel in terms of its implementation, it isn't intended to provide any functionality that does not exist. Given that there are existing JavaScript interpreters that differ only in that they do not provide the same guarantees about memory leaks and segfaults, our project has essentially no impact on privacy or ethical concerns.

Discussion

References

[1] Official Rust Language website https://www.rust-lang.org/

[2] Interview on Rust, a Systems Programming Language Developed by Mozilla http://www.infoq.com/news/2012/08/Interview-Rust

[3] LALRPOP, LR(1) parser generator for Rust https://github.com/nikomatsakis/lalrpop

[4] Mozilla Developer Network JavaScript Documentation https://developer.mozilla.org/en-US/docs/Web/JavaScript

[5] ECMAScript conformance test suite https://code.google.com/archive/p/sputniktests/