js.rs – A Rustic JavaScript Interpreter

CIS Department Senior Design 2015-2016 [1]

Terry Sun
terrysun@seas.upenn.edu

Sam Rossi
samrossi@seas.upenn.edu

April 25, 2016

Abstract

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.

We've built 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.

1 Background

1.1 Rust

Rust is a programming language spearheaded by Mozilla. It is a general-purpose programming language emphasizing memory safety and speed concerns. Rust's initial stable release (version 1.0) was released in March 2015.

Rust guarantees memory safety in a unique way compared to other commonly used programming languages. The compiler statically analyzes the source code, tracking the "lifetime" of any heap-allocated data. When all existant pointers to a piece of data has gone out of scope (e.g. at the end of a function), then the compiler determines that the data is no longer alive and the allocated space can be freed. (TODO: this simplifies a bit, do we want to go into more detail?) This memory management system prevents many classic memory errors by disallowing the programmer from accessing uninitialized or already-freed memory.

Compare this system to other memory-safe languages, which rely on garbage collection to find and free memory which is no longer relevant to the running program. Garbage collection incurs significant runtime performance costs.

Rust code compiles down to a native executable. The Rust compiler uses an LLVM-backend to emit assembly, taking advantage of LLVM's extensive optimization options. In addition, there is no runtime associated with executing Rust binaries; Rust is not run

[1] Advisor: Dr. Steve Zdancewic (stevez@cis.upenn.edu)

by a virtual machine (e.g. Java), nor does it use garbage collection during execution.

Rust also provides high-level programming language features, such as an type system based on interfaces (known as Traits) and generics, funtional programming and closures, and a set of concurrency primitives. This makes it a very attractive languages for people who are concerned about the performance of their program, but wish to use something more ergonomic than C or C++. The Rust standard library is written with both performance and ergonomics in mind.

1.2 JavaScript

JavaScript is an extremely widely used programming language. It is one of the core languages used on the Internet; it is included on the majority of the websites across the Internet, and is supported by all major web browsers. In addition, it has many desktop applications, e.g., used as a scripting language, or embedded in PDFs.

JavaScript has been deployed as a server-side language since late 1995. It has gained popularity as a server development language since Node.js was released in 2006.

TODO: Language features

JavaScript was first developed at Netscape by Brendan Eich 1995 for inclusion with their internet browser. Today, JavaScript is defined by the ECMAScript specification, first released in 1997 by the European Computer Manufacturer Association (ECMA) in response to the emergence of a few forks of the JavaScript implementations. The current version is ECMAScript 6, released in June 2015; ECMAScript 7 is still under development.

Mozilla Developer Network's official JavaScript documentation[4]. There is also a conformance test suite named Sputnik, released by Google in 2009. Sputnik targets the ECMAScript 5 standard, but includes only those features which were present in ECMAScript 3. This arrangement was chosen due to the presence of numerous ambiguities in the third specification, which were resolved in ECMAScript 5.

1.3 Motivation

TODO

2 Design

2.1 Parser

The first part of the interpreter is the parser, takes in a string of JavaScript and generates an Abstract Syntax Tree (AST). An AST contains the structure and the content of a program, but not the specific syntactic components such as whitespacing.

One very common way of writing a parser is to use a parser generator. This consists of defining the grammar of the language (i.e. the valid tokens of the language and the valid sequences of those tokens which have semantic meaning in the language); the parser generator then generates code to parse the grammar, which is then used as a library. Parser generators tend to be a bit slower in practice than writing an equivalent parser manually, but using a parser generate greatly increases the rate of development due to the ease of use. We opted to use a parser generator rather than writing a custom parser, as we wanted to focus on language implementation rather than performance.

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].

2.2 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.

2.3 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 we will refer to as the "runtime".

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.

3 Results

We implemented a significant subset of features which provide more than enough coverage to write interesting and useful programs.

3.1 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.

3.2 Specification 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 cate-

gories 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

Sputnik provides a total of 2427 distinct tests. We pass 20% (TODO) of those tests.

4 Ethics and Privacy

Although our project is novel in terms of its implementation, it isn't intended to provide any new functionality not present in existing solutions. Given that the 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 ethical concerns.

5 Discussion

LALRPOP errata

While LALRPOP generally worked quite well for our purposes, there were a few issues we ran into during the development process.

Compilation speed

While parser generators greatly facilitate development, they tend to be less performant than custom parsers specifically written for the source language. Although we were not heavily concerned with the code execution speed of our interpreter, we were fairly inconvenienced by how long it would take LALRPOP to generate our parser. As LALRPOP only had a single author who was quite busy with other things, we understood that we would not be able to expect as good performance as more mainstream parser generators. Given more time to improve js.rs, we likely would have implemented a custom parser to alleviate the issue. That being said, several non-optimization related updates to LALRPOP released during the course of our project greatly improved the development process, including the addition of human-readable error messages for shift-reduce and reduce-reduce conflicts.

Lexing

Unlike more mainstream parser generators, LALRPOP does not provide any built-in way to use a custom lexer. By default, LALRPOP will tokenize the input by splitting on any whitespace (and throwing out the whitespace itself). While this would work well in many scenarios, JavaScript interpreters typically allow newlines to be used in place of trailing semicolons. Additionally, correctly parsing single-line comments logically requires the detection of newline characters. In order to replicate this behavior as closely as possible with LALRPOP, we resorted to preprocessing the code run through js.rs before parsing it. Like the issue with compilation speed, we likely would have solved this by writing a custom parser if we had had enough time.

Multi-package architecture

One of the choices we made early on in the development of js.rs was to split the interpreter into multiple packages. Although this originally was intended to separate the code of French Press from the rest of the interpreter, soon decided to split out other parts of the interpreter as well. We ended up with four different packages, namely a "common" package with code needed by all other packages, French Press, the parser, and the runtime. Although this logical separation made sense in the abstract, it quickly became cumbersome to ensure that each crate's dependencies were in sync (e.g. that the parser and the runtime both used the same version of the common package). Moreover, due to the slow speed at which the parser

compiled, changes to the common package became rather undesirable, as each change would require the parser to recompile as well. In retrospect, using separate modules within a single package rather than :ultiple packages would have significantly increased our development efficiency.

6 Conclusion

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/