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.

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[1].

[1] https://github.com/nikomatsakis/lalrpop

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[2]. 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:

```javascript
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.

[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript

Results

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