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 mis-management.

Js.rs is 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 Introduction

1.1 Project

TODO: define scope of project

1.2 Motivation

TODO

2 Background

2.1 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.

[1] Advised by Dr. Zdancewic (stevez@cis.upenn.edu)

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 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.

2.2 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 is a high-level, imperative programming language. It is typically interpreted, and is loosely typed; any variable can be passed into any function or operator. The type system is composed of seven primitives types (Number, Boolean, `null`, `undefined`, String, Object, Symbol). Custom types can be created by inheriting from Object, using a prototype-based inheritance system. JavaScript has many function features, such as first class functions, anonymous functions, and closures.

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.

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: Mozilla Developer Network's official JavaScript documentation[3]. 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.

3 Design

3.1 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.

We wrote our parser using a parser generator, a common pattern in building parsers. 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 takes this grammar specification as input and generates code to parse the grammar in the target language (in this case, Rust). 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 maintain the safety guarantees. After some research, we decided to use LALRPOP, a pure Rust LR(1) parser generator[2].

3.2 Garbage Collector

JavaScript is a garbage collected language, as it relies on a runtime system to manage the memory used by a running program. This runtime must allocate memory when requested by the language, and free memory when it is determined that allocated data is no longer accessable by the running program. We used a garbage collection library, French Press. French Press was being developed concurrency by David Mally, a UPenn Masters student, for his Masters thesis. Js.rs is tightly coupled with French Press, sharing a common library defining a shared representation of the JavaScript type system.

Type System

In JavaScript, a value can either be located on the stack or on the heap. Primitive values (e.g. numbers, boolean) are located on the stack, whereas as all references (such as objects or anything created from a constructor) are located on the heap. To model this properly, our type system used a pair of types for each value. `JsVar` contains the type of the value, as well as any stack-related value (such as the Rust number or boolean containing the value itself); additionally, each `JsVar` can optionally be paired with a `JsPtrEnum`, which contains a reference to any heap allocated data that the value may require. Each `JsVar` also contains an optional unique binding, which acts as a lookup identifier for use with heap datatype provided by French Press

For example, for the object { `x: 3.5, y: "foo"` }, the `JsVar` would indicate that the value is of type "object". The `JsPtrEnum` for the object would contain a `HashMap` with the keys `"x"` and `"y"`. The value for `"x"` would be a `JsVar` of type "number" and with floating point value `3.5`, and the value for `"y"` would be a `JsVar` with the type "string" and a unique binding which can be used to look up the heap data.

3.3 Runtime

The "runtime" is the most substantial portion of the interpreter, as it is what accepts the AST as input and evaluates the language semantics represented by each node. TODO

Functions, Scoping, and Closures

One of the most complex 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.

4 Results

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

4.1 Language Features

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

Assignment statements

There are two ways to assign values to variable bindings in JavaScript. A declaration (`var x = y`) will create a variable in the current scope and assign the value y to it. An assignment (`x = y`) will modify the variable x in the current scope or any parent scope if it already exists; otherwise, it will be allocated in the global program scope and set to the value y. Additionally, a variable can be declared but not set (`var x`), which sets it to the default value `undefined`.

Literal expressions

Literal expressions evaluate to the value they represent, and are defined in JavaScript for each of the primitive types. For `null`, `undefined`, boolean, and numeric literals, we simply create a variable containing the appropriate value. For more complex literals which must be allocated on the heap (Strings, Objects, and Arrays), we first evaluate the expression by constructing the appropriate type. For Strings, this includes parsing escaped values and translating escaped Unicode values to their corresponding Unicode characters. For Objects and Arrays, each sub-component in theliteral must be iteratively parsed, and the value constructed. Then, we allocate heap space and store the value.

- Boolean: `true`, `false`
- Numeric: e.g. `-4`, `7.17`, `NaN`
- `null`
- `undefined`
- String: e.g. `"abc\u1234"`, `'foo bar baz'`
- Object: e.g. { `x: 3, y:` { `z: "hello"` } }
- Array: e.g. `[1, "hello", x]`

Operator expressions

JavaScript provides a set of binary (`a + b`) and unary operators (`++a`) to define expressions based on one or more variables. JavaScript does not enforce type bounds at all, so Js.rs must perform type coercion when an expression contains values of two different types. For example, in a logical and statement, both expressions are first coerced to an boolean value before the logical and is applied.

This may involve operator overloading, as in the case of `+`: in most cases, the interpreter must coerce both arguments to Number before evaluating the expression. However, if either argument is a String, then both arguments should instead be coerced to Strings and the operation results in the concatentation of those two Strings.

Additionally, all binary operators can be used as assignment operators (e.g., `a += b`. This application in Js.rs is handled by the parser, which will produce an AST node identical to the `a = a + b` case.

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

Function-related expressions

Functions are stored into the current scope as a special meta-level type which contains the function name and the AST node for the function body. When a function is called, the interpreter allocates a new scope as a child of the current scope, and will then execute the semantics stored in the function's AST node.

As discussed above, anonymous functions may contain references to its parent scope, creating a closure environment.

- 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

Js.rs supports simple Object functionality as a key-value store. Js.rs does not support the Prototype inheritance model.

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

Control-flow statements

Many JavaScript constructions change the execution flow of the program. Typically, the interpretere will linearly execute the AST nodes as returned by the parser. However, the interpreter may traverse into previously encountered nodes (e.g., a function call), repeat the current node until a condition is met (e.g., for and while loops), or skip certain AST nodes (e.g., an if statement).

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

Standard library

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

Printing

JavaScript interpreters typically provide a global `console` object, through which programs can accept input and provide output to the developer console. We implemented an abbreviated `console` containing only the `log` function, which coerces its first argument to a string value and outputs it to the screen.

Prototypes

JavaScript packages a number of built-in prototypes, such as `String`, `Array`, `Object`, and `Function`. A value of a given prototype can be created with the `new` keyword, e.g., `new Object()`. Js.rs implements many of these prototypes as simply wrapper types around the primitive `String`, `Number`, and `Boolean` types. We implemented the `new` keyword by executing a call to the appropriate constructor.

Additionally, Js.rs packages a basic `Array` prototype, with support for constructing Array objects from literals, `push`, accessing elements by index, and re-sizing by setting the length property.

4.2 Specification Coverage

To test how complete our coverage of the JavaScript standard was, we built a framework to run the Google Sputnik test suite[4] on our interpreter. 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. 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).

We considered the number of leaf categories in which we had passed some of the tests. Of the 111 categories, we had coverage in 73 of the categories, or 65.8%. This indicates that we covered a sizable portion of the languages features.

Raw coverage

Sputnik provides a total of 2427 distinct tests. In the end, Js.rs passes 18.2% of those tests. Despite having implemented a large subset of the commonly used JavaScript features, there were some features that we did not have time to implement. Some of these included:

- The global `Arguments` object
- The conditional operator (`?  :`)
- The `void` operator
- The `delete` operator
- The `in` operator
- The `with` statement

5 Ethics and Privacy

Js.rs is developed as a free and open source software project, hosted on GitHub. Potential users do not have to blindly trust that the interpreter is not malicious, but can audit it for themselves. As running this project on a JavaScript program would require revealing the full source to the interpreter, it is important that Js.rs does not contain any information exfiltration systems or other backdoors.

6 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 c̈ommonp̈ackage 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 multiple packages would have significantly increased our development efficiency.

7 Conclusion

TODO

References

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

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

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

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