Similar Function Elimination

Table of Contents

Introduction

This document covers the design and implementation details of Similar Function Elimination, a proposed Emscripten optimization pass that has the potential to reduce Emscripten's asm.js payload size.

Motivation

If we were to eyeball the asm.js generated by Emscripten today for template heavy codebases, it immediately becomes obvious that there are several generated functions in the asm.js that are structurally very similar, yet not being optimized into helper functions that can be invoked from the original function. Examples of such functions will be presented in detail below, but for now, suffice it to say that a merging of such functions presents an opportunity to reduce the size of our JavaScript payload significantly, helping us meet our performance goals.

Background Reading

It is expected that the audience is familiar with the following:

a. Emscripten – Transpiler tool that we use to convert our C++ into asm.js JavaScript.

b. asm.js – A subset of JavaScript that modern browsers can compile ahead of time (AOT) to reap performance benefits.

c. NodeJS – Runtime based on the V8 JavaScript engine

d. Abstract Syntax Tree – A tree representation of the source code for a program that is typically generated by a parser.

e. Esprima - Esprima is a high performance and lightweight JavaScript parser that allows for fast parsing of JavaScript files. Existing tools used by Emscripten such as UglifyJS can become pretty unstable when parsing through large Javascript files (specifically, they hit the garbage collection limit of ~1.4 Gb that is imposed by NodeJS and this results in thrashing).

   Esprima is now maintained by the JQuery foundation.

f. Escodegen, Estraverse - Escodegen and Estraverse are tools associated with Esprima – specifically, escodegen allows you to generate code from an Esprima parsed AST (which obeys the ESTree spec), and estraverse allows for quick traversal of an Esprima parsed AST.

g. Mocha, Chai – Mocha and Chai are two JavaScript libraries that are used for testing of production NodeJS JavaScript. Mocha is a test framework while Chai is an assertion library.

Design Goals

The following design goals have been envisioned for the Similar Function Elimination (SFE) feature:

It should run fast

SFE should run as fast as possible – we should make use of the latest and fastest JavaScript parsers and technologies to ensure that the build time overhead for the end developer is minimal. This is in line with our stated goal of improving developer productivity.

It should be performant

SFE should optimize as much as possible – wherever possible, we will optimize for code size reduction. Said another way, if we were to run another pass of SFE on an SFE reduced file, we should not be able to reduce the file any further

It should be stable

Emscripten can generate some pretty large JS files – some files such as core-test.js optimized are currently at over 10Mb. SFE should be able to handle such files without any issues.

It should generate valid asm.js

Emscriptenized files conform to the asm.js standard. The output of SFE should also conform to the same standard.

Examples

With these design goals stated, let us consider a few examples of JavaScript function types that will benefit from Similar Function Elimination. Keep in mind that all of the presented examples will be using generated and optimized asm.js code. We will use the examples to inform our design.

Example 1 – Two functions that differ by a single literal

```
function XVa(a, b) {
    a = a | 0;
    b = b | 0;
    return (ZR(c[a + 24 >> 2] | 0, b) | 0) != 0 | 0
}

function z2a(a, b) {
    a = a | 0;
    b = b | 0;
    return (ZR(c[a + 12 >> 2] | 0, b) | 0) != 0 | 0
}
```

It's pretty obvious by looking at the two functions above that they can be coalesced into a helper function that takes a, b and a third literal parameter as input. Here's the transformation that we shoot for with SFE:

```
function XVa(a, b) {
    return zzc(a, b, FP, 24)
}

function z2a(a, b) {
    return zzc(a, b, ZR, 12)
}


function zzc(a, b, zzd, zze) {
    a = a | 0;
    b = b | 0;
    return (zzd(c[a + zze >> 2] | 0, b) | 0) != 0 | 0
}
```

Example 2 – Two functions that differ by a single identifier

Consider the next example in which two functions differ by a single identifier.

```
function a(d, e, f)
{
    var abc = 0;
    abc = abc+1;
    abc = abc*c(d, e);
    return;
}

function b(d, e, f)
{
    var abc = 0;
    abc = abc+1;
    abc = abc*g(d, e);
    return;
}
```

Here is what the reduced code could look like – we identify that a and b are exactly the same function except for the identifiers c and d being invoked in each function respectively. We thus hoist the common code into a helper that takes the identifier as an additional parameter.

```
function a(d, e, f) {
    return h(d, e, f, 0)
}

function b(d, e, f) {
    return h(d, e, f, 1)
}


function h(d, e, f, i) {
    var abc = 0;
    abc = abc + 1;
    abc = abc * table[i&1](d, e);
    return
}

var table = [c, g];
```

In the above example, we identified that the functions a and b were identical except for the functions c and d that they were invoking. The asm.js standard does not allow for passing string identifiers as arguments. It only allows for int and float types. It does however allow for specification of function tables at global scope. We thus create a new table for the functions c and g and pass in the indices of c and g as parameters to the newly generated function h.

Example 3 – Multiple Functions Sharing Structural Similarity

This example illustrates what SFE could do in the general case.

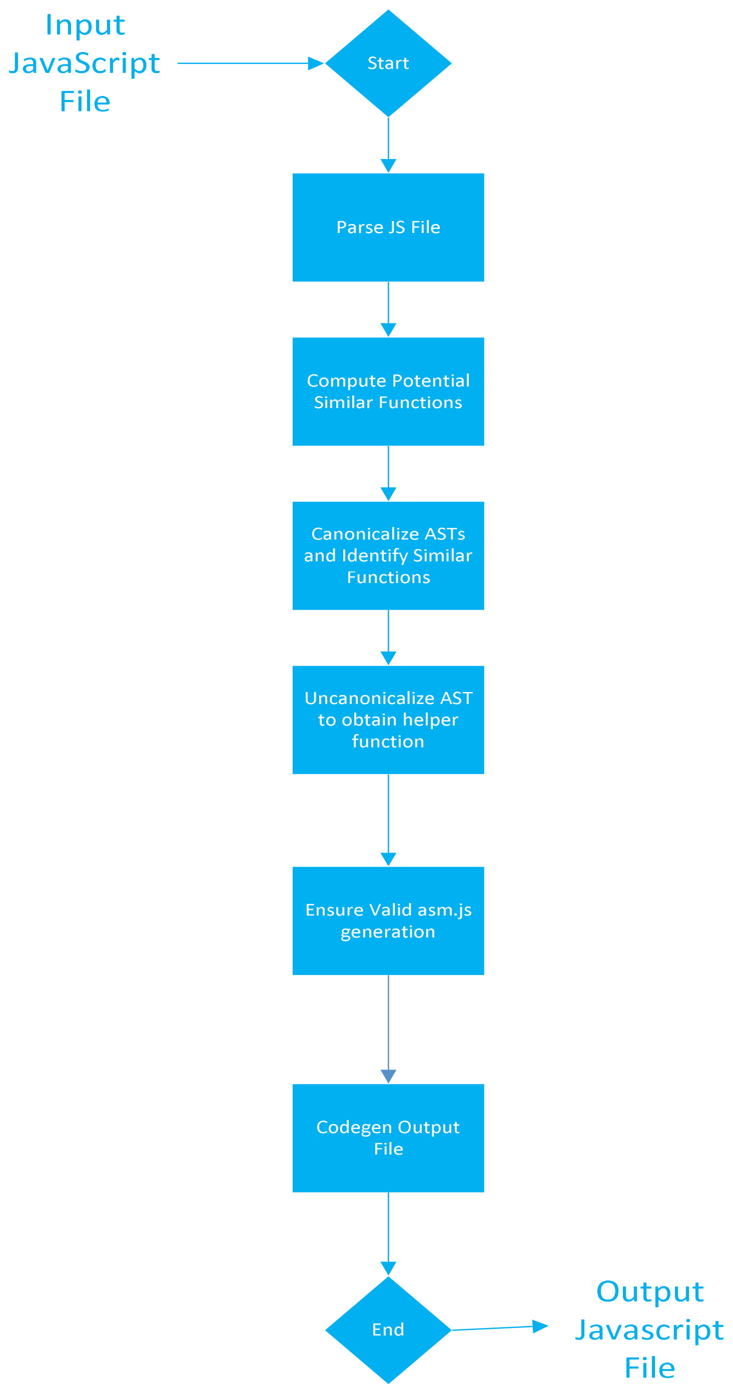
```
function njb(a, b) {
    a = a | 0;
    b = b | 0;
    xc(a | 0, 1, 25488, 419075, 2633, b | 0);
    return
}

function anb(a, b) {
    a = a | 0;
    b = b | 0;
    xc(a | 0, 2, 25376, 417662, 904, b | 0);
    return
}

function Fnb(a, b) {
    a = a | 0;
    b = b | 0;
    xc(a | 0, 4, 25472, 293066, 46, b | 0);
    return
}

function Gqb(a, b) {
    a = a | 0;
    b = b | 0;
    xc(a | 0, 2, 25460, 417662, 912, b | 0);
    return
}
```

```
function njb(a, b) {
    return $$$e(a, b, 1, 25488, 419075, 2633)
}

function anb(a, b) {
    return $$$e(a, b, 2, 25376, 417662, 904)
}


function Fnb(a, b) {
    return $$$e(a, b, 4, 25472, 293066, 46)
}

function Gqb(a, b) {
    return $$$e(a, b, 2, 25460, 417662, 912)
}

function $$$e(a, b, $$$f, $$$g, $$$h, $$$i) {
    a = a | 0;
    b = b | 0;
    xc(a | 0, $$$f, $$$g, $$$h, $$$i, b | 0);
    return
}
```

Multiple functions sharing structural similarity were identified and a helper was generated that could coalesce all of them.

High Level Design

At a high level, the similar function algorithm takes in a JavaScript file, parses it to generate the AST for the file, computes the potential similar functions by identifying functions with common formal lists and number of lines, then canonicalizes the ASTs of the potential similar functions and hashes their bodies to identify similar functions, uncanonicalizes an exemplar AST from each set of similar functions to obtain the helper function AST, and finally reduces each AST in each set of similar functions to invoke the appropriate helper function. The follow flow diagram illustrates this.


Input JavaScript File
Start
Parse JS File
Compute Potential Similar Functions
Canonicalize ASTs and Identify Similar Functions
Uncanonicalize AST to obtain helper function
Ensure Valid asm.js generation
Codegen Output File
End
Output Javascript File

We now consider each of these steps in detail.

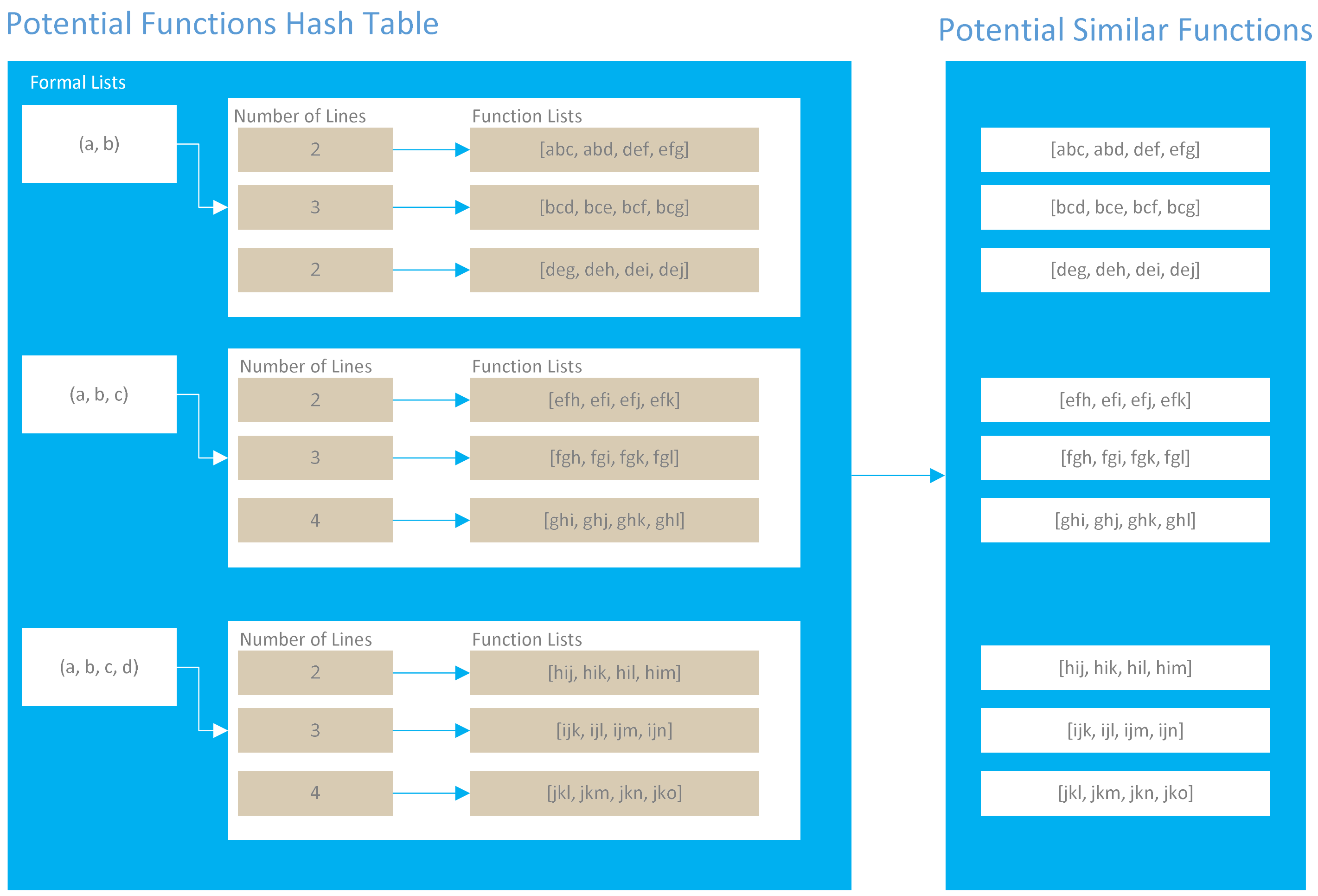
Computing Potential Similar Functions

A naïve approach to similar function elimination might end up comparing each function with every other function. This would be prohibitively expensive since Similar Function Elimination uses AST comparison (discussed shortly) to identify similarities in functions. The AST for even a very simple function contains a decent number of nodes and an AST comparison function typically takes O(n) time where n is the number of nodes in the tree. Thus, in order to reduce the running time of the algorithm, we prune the sets of functions that can be compared by creating a two way hash table:

a. The first level buckets functions based on their formal lists (a.k.a. parameter lists). In the examples section, we had considered functions with the same formal list namely **(a, b)**.

b. The second level buckets functions based on their number of lines. Since for similar function elimination, we are only considering functions that have the same structure, we can afford to bucket further on the number of lines to identify potential similar functions.

Once we compute this two way hash table, we then aggregate the lists of functions that share the same formal parameters and line numbers into a nested list of potential similar functions. The following diagram illustrates this aggregation.



Canonicalization

The next thing we do is canonicalize the ASTs of the potential similar functions that we have identified. Canonicalization involves a few steps:

a. We identify all function identifiers, variable identifiers and literal identifiers in the ASTs of the identified potential similar functions
b. We replace these identifiers with placeholder values
c. We hash the body of the resultant AST
d. We use this hash to compute similar functions. Functions with the same hash are identified as similar.

We append the newly generated helper function's AST to the end of the AST for the program.

Uncanonicalization

Uncanonicalization involves creating a copy of one of the ASTs in each set of similar functions. We traverse the AST, generating identifiers for each literal/identifier that was identified as a placeholder in step b above.

Some things to keep in mind:

a. We use indices of function identifiers along with their corresponding function table for call expressions.
b. We add parameter annotations to parameters to conform to the asm.js spec. Parameter annotations are used to statically denote the underlying data type to the ahead-of-time compiler used by the JS engine.

Ensuring Valid asm.js Generation

In order to generate valid asm.js, we need to ensure that we do not violate asm.js restrictions when modifying the ASTs of similar functions and when generating helper functions.

Some examples:

a. Ensuring that we properly shift (sometimes by 0) when passing identifiers (that were previously literals) as indexers into typed arrays.
b. Ensuring that we invoke Math.imul when converting literals to identifiers when the * operator is used
c. Adding appropriate annotations to identifiers (that were previously literals) in binary expressions, unary expressions, etc.
d. Ensuring that we don't add functions that are not present to function tables
e. Ensuring that we make sure that we use bit masks to ensure that we don’t overflow the bounds of function tables when indexing into them. In order to enforce this, each function table that we generate has to have a length that is a power of 2.

In order to enforce these checks and keep them decoupled from the core SFE algorithm, we mark each placeholder value and push it onto a queue when uncanonicalizing. We then subsequently traverse the queue post uncanonicalization and perform the necessary AST modifications to enforce valid asm.js.

Code Generation

We use escodegen to traverse the AST and generate the code for our reduced source file.

Further Work

Parallelization

In order to further reduce the running time of Similar Function Elimination, we should be able to develop a parallel version of the algorithm. Note that once we’ve identified potential sets of similar functions, we can divide up the work of identifying merge candidates and perform AST reduction across multiple cores.

Relaxing AST Diff Constraints

Another area to explore is to relax the constraints on structural similarity. The current proposed version of SFE only works on functions whose ASTs are exactly the same except for differing literal values or identifier names. Consider the following example.

```
function a(d, e, f)
{
    var abc = 0;
    abc = abc+1;
    abc = abc*3;
    return;
}

function b(d, e, f)
{
    var abc = 0;
    abc = abc+1;
    abc = abc*g();
    return;
}
```

The current version of SFE will not work on a and b above since it requires structural equivalence. In this case, b invokes an identifier, but a uses a literal on the 4th line. In theory, we should be able to generate a helper function as follows:

```
function a(d, e, f)
{
    return $$$_(3);
}

function b(d, e, f)
{
    return $$$_(g());
}

function $$$_(h)
{
    var abc = 0;
    abc = abc+1;
    abc = abc*h;
    return;
}
```

This type of transformation merits further investigation, but will need further work on the AST comparison algorithm and the process of generating helper functions, etc.