Institiúid Teicneolaíochta Cheatharlach

INSTITUTE of TECHNOLOGY CARLOW

At the Heart of South Leinster

Design Manual

11/01/2013

Daniel Connor

C00137906

Table of Contents

Introduction

JavaScript[1] is a dynamic language created originally in 1995 by Brendan Eich for the netscape browser. Its primary use is in the browser to make web pages interactive, although recently it has also become popular on the server side through javascript engines such as rhino[2] and node.js[3]. As javascript has become more popular there is a greater need for tools to help developers test their code. Projects such as phantom.js[4], which is a “headless browser”, allows the browser to be scripted using javascript. There are also static analysis lint tools[5, 6] that analyse the code for syntax errors or reference errors. This has proven to be useful in allowing browser testing to be automated. However there aren’t many automated tools that create test cases. The ones that do are mainly oriented around security and vulnerabilities associated with javascript[7, 8] and do not focus on the quality of javascript code, or the amount of errors that occur.

Runtime errors should not occur in production software. What we want to do is to generate runtime errors in javascript to highlight places where error checking should be improved before the code is released. It should also help to highlight places where a programmer is misusing a API, whether that is a native API provided by the environment in which the javascript is running, or a third party API such as jQuery[9], Backbone[10], or underscore[11]. If we cannot cause any, it should prove that the software does enough error checking to be stable and uses any APIs correctly.

The DOM(Document Object Model) API is the api that is provided by browsers to allow javascript to manipulate a web page. The DOM is made up of a tree of Nodes, which are extended to provide different kinds of functionality to a web page. Because javascript the main use of javascript is in the browser, and it’s primary use there is to manipulate web pages to allow them to be interactive, that is where we are going to concentrate our efforts.

I have decided to name the tool JSSeek as its purpose is to seek out errors in javascript code. The tool will be referred to by this name throughout the rest of this document.

Since javascript is a dynamic language, we need to handle both the types of values and also the values of variables in the code. For example, in a statically typed language an integer will always represent an integer value, however in javascript a variable can contain any type. For example when a variable is passed into the function we cannot know what type of variable it is, so it could contain a number, a string, an object or a boolean value. Basically it can contain any type in the language. Now in a language like c, numbers cannot have methods. In javascript almost any value can have methods on their prototype. There are only two values that will throw an error if a method property is accessed from them; null and undefined. Other errors may occur if say a number is tried to be invoked.

The main language we are going to use is javascript. The reasons are as follows:

- Easy to run from the command line using node.js and allows the possibility of running the tool in the browser
- Easy access to javascript specific operators such as === and == (these are handled differently than other languages) so we don't have to define rules.
- Ability to access DOM api from any javascript environment.
- Availability of a module to parse javascript into an AST(Abstract Syntax Tree)

Javascript Subset

Due to the fact that we are merely doing a proof of concept, we will only take into account a subset of javascript, as the work involved to include every part of javascript would be too much work for the limited time. Here we define the exact subset of javascript that we will aim to include:

Including

- switch statements
- If statements - These are the main flow of control statements so without them our tool would be useless.
- Loop statements - While we may not be able to handle loop statements fully, we will aim to at least handle them in a simple manner.
- Operators - including instanceof, in, typeof, delete and all other boolean and arithmetic operators.
- Function calls - We will handle passing arguments to functions and also the scope of functions including the this keyword.
- Prototypal inheritance
- We will aim to include most of the new Object methods introduced in Ecmascript 5 [15]

Excluding

- Regular expressions
- String Manipulations - Due to the nature of strings, we will not be handling their modification, we will however handle strings with constant values.
- We won't be including support for new Array methods introduced in Ecmascript 5 [15]
- We won't be including any support for new features that will be introduced in Ecmascript 6 [14]

These are some samples of javascript from underscore.js[11] that we should will aim to be able to handle

```
_.bind = function(func, context) {
    var args, bound;
    if (func.bind === nativeBind && nativeBind) return nativeBind.apply(func,
slice.call(arguments, 1));
    if (!_.isFunction(func)) throw new TypeError;
    args = slice.call(arguments, 2);
    return bound = function() {
      if (!(this instanceof bound)) return func.apply(context,
args.concat(slice.call(arguments)));
      ctor.prototype = func.prototype;
      var self = new ctor;
      ctor.prototype = null;
      var result = func.apply(self, args.concat(slice.call(arguments)));
      if (Object(result) === result) return result;
      return self;
    };
  };

_.pick = function(obj) {
    var copy = {};
    var keys = concat.apply(ArrayProto, slice.call(arguments, 1));
    each(keys, function(key) {
      if (key in obj) copy[key] = obj[key];
    });
    return copy;
  };

 _.extend = function(obj) {
    each(slice.call(arguments, 1), function(source) {
      if (source) {
        for (var prop in source) {
          obj[prop] = source[prop];
        }
      }
    });
    return obj;
  };
```

Analysis

Parsing

To parse the javascript code, we are going to use a module called esprima. The module can parse any piece of javascript and gives an abstract syntax tree(AST) as a result. The AST conforms to the mozilla parser API[13].

Program State

The program state manages the possible values and scopes that exist as the function is executed.

Scope - to handle scope we need to have the concept of a scope in which variables are defined.
FunctionScope - In javascript a scope is only created for every function, not every block as in c.
stack - The function scope's stack is an array of the possible states of the stack in the function.
We need to emulate javascript scope.

Values

We will use a number of different constructs to represent the values of a variable.

VagueValue

A vague value represents a value that is still "vague" to us. We do not know its exact value but we may have some constraints that define what kind of value it may represent. The constraints are built up as each constraint scope is entered.

The types that inherit from this type are defined below:

VagueNumber

As we build up constraints that define a number they will get more and more complex. Also, we need to be able to perform operations on a vague type that will modify its value which further complicates things. We could have a representation of a value such as this:

```
A>45 and B-5=A*A
```

When we reach a constraint involving a vague number we need to be able to figure out whether the number satisfies the constraint. e.g. in this case we need to figure out whether 11 in the range of values allowed by the constraint above.

```
if ( X == 11 )
```

To solve these constraints we will use ptc solver. This library allows us to check whether a particular constraint allows any values.

VagueObject

The VagueObject type represents a javascript object that we do not know the attributes of. The constraints of an object differ a lot from the constraints that are applied to numbers. If we access a property of an object that we do not know for certain that it has, then the value of that property can be anything. Constraints on object values allow us to define what properties the object has.

A vague object can have symbolic properties set on it because it has similar properties to a hash map. Because a symbolic property is essentially a range of values and we do not know which of the values is set, we add the value as another option to each of the values allowed by the constraint.

```
if(obj.prop) {
  // obj.prop can be any truthy value
  // obj.anyOtherProp can be any value
}
```

VagueArray
An array inherits from an object but has functions that allows it to mimic the behaviour of a real javascript array.

VagueFunction
A function is essentially the same as an object with the property that it is invokable. In the case that we know that we have a function, but we do not know what it is, we make the assumption that the function can return anything. This is a reasonably safe assumption to make because if the function is passed in as an argument, then the function can indeed return any value.

VagueBoolean
A vague boolean type represents a condition that cannot be evaluated. For example when a variable assigned a value such as the following example where ‘a’ is a VagueNumber.

```
var x = a < 10;
```

ConcreteValue:
Concrete types are used when we have a variable that we know the exact value. Operations performed on vague types are recorded for later use, however for concrete types we can apply the operations the the values as would happen during normal execution.

There are two cases in which we get concrete values.

- If a constraint limits a VagueType to one value. e.g. if the constraint is x == 1. Then there is no doubt that x has a value of 1 within the scope of this constraint.
- If a variable is assigned to be a concrete value. This applies to every assignment statement that is reached.

Special types
The following types are represented as concrete types:

`undefined` - This is a constant and represents it’s own type.
`null` - Although `typeof null` yields “object” it does not possess the same properties as an object.
`NaN` - This has some of the same properties as a number, but has some special properties.

ValueRange

To handle the multiple values that a variable may have at any point in a program’s execution we use a value range. A value range acts like an set. When a constraint is applied to the ValueRange, the array is filtered using the constraint to represent all the values that satisfy the constraint. A value range will only ever contain one instance of any ConcreteType.

Operations that are performed on a value range must be applied to each value in the range. More on this is explained in the Operations section.

Anything

When we begin analysing a function, the arguments can contain any possible value in javascript. For this reason we represent each of the arguments as a ValueRange containing all possible options. This is the basis for our implementation. If the constraints imposed on the inputs allow an invalid operation to be performed on an object then we have then we know an error will occur.

```
VagueBoolean,
VagueNumber,
VagueObject,
String,
null,
undefined,
NaN,
Infinity
```

Operations

Due to the fact that javascript is a dynamically typed language, standard operators that would normally be used in combination with numbers can be used on other types. Now because a variable may have multiple values, we must apply the operation to all the values. In the case of a binary operation, if both operands are vague types, then the operation needs to be applied to every combination of values. If the result of the operation between two operands is the same as another, then we only include the result once as long as the result is a scalar value.

Here are some samples of how operations between different types work:

```
function(a) {
  return a + 1;
}
```

Operand 1	Operand 2	Result
VagueBoolean()	1	b + 1
VagueNumber()	1	x + 1
VagueObject()	1	NaN
null	1	1
undefined	1	NaN
Infinity	1	Infinity
NaN	1	NaN

```
function double(a) {
  return a + a;
}
```

Operand 1	Operand 2	Result
VagueBoolean()	true	b + b
VagueNumber()	VagueNumber()	x + x
VagueObject()	VagueObject()	"[object Object][object Object]"
null	null	0
undefined	undefined	NaN
Infinity	Infinity	NaN
NaN	NaN	NaN

Code

If Statements

Each if statement that is encountered acts as a constraint for each of the values that it checks. There are some examples in the section on types above. What this means is that the if statement limits the possible values of a variable throughout the if statement.

While within a function, there must exist at least one set of possible values for each of the variables, what we will call a state.

At each `if` statement, there will be one or more states. When a state is created, the branch that caused the set to exist is recorded with the set which allows us to identify the Path Traversal Condition PTC. Each of the states are checked to see whether it can pass the constraint imposed by the if statement. If a possible set of values will always pass the constraint, then the state itself is modified in the if statement rather than creating a new one. Conversely if the values in the set will always *not pass* the constraint, then the set remains as is. If this is the case and there is an if branch, the set will be modified in the else branch.

At `else` statements the constraint in the if statement is inverted and the same process applied to each of the states.

Here we present an example that shows the states that are created during function execution. Note however that we have a simplified representation of the states: Each of the variables do not show all possible values.

Line	Code	Options
1	`function sample(a) {`	
2	`var b;`	`[ { constraint: null, b: [ undefined ], a: [ VagueNumber(), VagueBoolean(), Infinity, NaN, null ] } ]`
3	`if(a > 0 && a < 10) {`	`[ {`

```
                                         pathCondition: a > 0 && a < 10,
                                         b: [
                                           ConcreteValue(undefined)
                                         ],
                                         a: [
                                           VagueObject(),
                                           VagueNumber(),
                                           VagueBoolean(),
                                           ConcreteValue(Infinity),
                                           NaN,
                                           null
                                         ]
                                       }
                                     ]
4        b = true;                   [
                                       {
                                         pathCondition: a > 0 && a < 10,
                                         b: [
                                           ConcreteValue(true)
                                         ],
                                         a: [
                                           VagueNumber(),
                                           VagueBoolean(),
                                           Infinity,
                                         ]
                                       }
                                     ]
5      }
6      else {                        {
                                         pathCondition: a <= 0 || a >= 10,
                                         b: [
                                           ConcreteValue(undefined)
                                         ],
                                         a: [
                                           VagueObject(),
                                           VagueNumber(),
                                           VagueBoolean(),
                                           ConcreteValue(Infinity),
                                           ConcreteValue(NaN),
                                           ConcreteValue(null)
                                         ]
                                       }
                                     ]
7        b = false;                  {
                                         pathCondition: a <= 0 || a >= 10,
                                         b: [
                                           ConcreteValue(false)
                                         ],
                                         a: [
                                        VagueObject(),
                                        VagueNumber(),
                                        VagueBoolean(),
                                        ConcreteValue(Infinity),
                                        ConcreteValue(NaN),
                                        ConcreteValue(null)
                                      ]
                                    }
                                  ]
 8      }
 9                                [
                                    {
                                      pathCondition: a > 0 && a < 10,
                                      b: [
                                        ConcreteValue(true)
                                      ],
                                      a: [
                                        VagueNumber(),
                                        VagueBoolean(),
                                        ConcreteValue(Infinity),
                                      ]
                                    },
                                    {
                                      pathCondition: a <= 0 || a >= 10,
                                      b: [
                                        ConcreteValue(false)
                                      ],
                                      a: [
                                        VagueNumber(),
                                        VagueBoolean(),
                                        ConcreteValue(Infinity),
                                        ConcreteValue(NaN),
                                        ConcreteValue(null)
                                      ]
                                    }
                                  ]
10      if(b) {                   [
                                    {
                                      pathCondition: a > 0 && a < 10,
                                      b: [
                                        ConcreteValue(true)
                                      ],
                                      a: [
                                        VagueNumber(),
                                        VagueBoolean(),
                                        ConcreteValue(Infinity),
                                      ]
                                    }
                                  ]
```

```
11      a += 10;                [
                                  {
                                    pathCondition: a > 0 && a < 10,
                                    b: [
                                      ConcreteValue(true)
                                    ],
                                    a: [
                                      VagueNumber(a + 10),
                                      VagueBoolean(),
                                      ConcreteValue(Infinity),
                                    ]
                                  }
                                ]
12    }
13                              [
                                  {
                                    pathCondition: a > 0 && a < 10,
                                    b: [
                                      ConcreteValue(true)
                                    ],
                                    a: [
                                      VagueNumber(a + 10),
                                      VagueBoolean(),
                                      ConcreteValue(Infinity),
                                    ]
                                  },
                                  {
                                    pathCondition: a <= 0 || a >= 10,
                                    b: [
                                      ConcreteValue(false)
                                    ],
                                    a: [
                                      VagueNumber(),
                                      VagueBoolean(),
                                      ConcreteValue(Infinity),
                                      ConcreteValue(NaN),
                                      ConcreteValue(null)
                                    ]
                                  }
                                ]
14    return a;
15  }
```

Line	Description
2	b has not yet been assigned a value and a can be anything as it was passed as an argument
3	The values get filtered to be by the constraint in the if statement

4	b is set to true
6	The original values are filtered by the negation of the constraint and an new option is added to the set of possible values.
7	b in the current set of possible values gets set to false
9	The current set of values at this point in the function are shown.
10	Only one state out of the two created at the last if statement is allowed by the constraint
11	The state that is allowed will always pass the constraint, so in this case we can update the state directly rather than creating a new one.

Otherwise if some of the values in the set allow the if branch to be taken, then a new possible set of values is created for the if branch which will be filtered using the constraint, the original set is filtered by the negation of the constraint. If an else branch exists in this case, the filtered-original set will be used as the set of values within the if statement.

Loops

Loops are the most complex form of flow of control statement as we can never know how many times it will be executed when symbolic values are involved. To make it simpler for the amount of time that we have, we will use a common solution to this problem. When we encounter a for loop, we create two new states. One state will represent the values if the for loop was not executed, the other state will represent the state if the loop was run a random number of times.

Path Values

Each path that we take will normally have a different set of values. For this reason the possible options are contained in a data-structure. If a variable contains a vague type, then operations need to be performed on each of the options related to paths as well as the possible values of a vague type. Each path option is associated with the constraint that it has to pass to exist.

Type Errors

As we trace through the code and build up a database on the various values of the variables, we reach operations that happen on values. Now we must use a combination of the data we have collected about each variable in combination with the operation to see if any of the possible values of the variable in question can cause a type error.

Here we have an example of the states that are currently available in a particular point in the code.
The next statement to be executed is `var b = a.property`.

```
states = [
  {
    pathCondition: a != null,
    a: [
      VagueObject(),
      VagueFunction(),
      VagueNumber(),
      ConcreteValue(NaN),
      ConcreteValue(Infinity)
    ]
  },
  {
    pathCondition: a == null,
    a: [
      ConcreteValue(null),
      ConcreteValue(undefined)
    ]
  }
];
```

As you can see we have two possible states at this point, but also that we have a further two options for each of these values. For each of the states we have to check if the operation is supported on that value. You can also see from the table below that the values for “a” in the first state will not throw an error because the operation is allowed. However in the second state the values are null and undefined. These values do not allow property access and will therefore throw a TypeError. Now that we know that it is possible for this operation to throw an error, if the path traversal condition(which is based on path traversal conditions) is feasible we generate a solution(we take a sample) using a constraint solver (e.g) the ptc solver. The code when executed using this solution will always generate a runtime error.

Value	Allowed
VagueObject	true
VageFunction	true
null	false
undefined	false

Property access allowed

Custom Errors

Other than type errors in a program, we need to check if a custom error will be thrown, now when we are analyzing a particular function, and we come across a piece of code like this:

```
if(x == 0) {
   throw new Error(“Division by zero error”);
}
```

We don’t need to generate an error message in this instance. When we do need to generate an error, however, is when this function gets called from another location where we know multiple values can be represented by the inputs, one of which meets the constraint imposed around the throw statement. Take this example:

```
function divide(x, y) {
  if(y === 0) {
    throw new Error(“division by zero error”);
  }
  return x / y;
}

function example(x) {
  var y = 0;

  if(x > 10) {
    y = 10;
  }
  else if(x <= 5) {
    y = 5;
  }

  var a = divide(x, y);
}
```

In the above example we need to show that by calling the function divide, that y can be zero and so can throw an error.

DOM Errors

To handle DOM based errors we need to define when errors will occur as we do not have access to the implementation. As again we are only doing a proof of concept we will only define some of the common functions. However the definitions can be easily expanded to include a more fully featured subset of the DOM API.

To generate errors we assume that we have no knowledge of the initial state of the DOM. This allows us to always assume that for example the return value of any selector statement can be `null` i.e. it doesn’t exist in the DOM. This may yield some false positives in some cases, so we will allow the DOM to be specified

Generating Inputs

strings - We don’t take symbolic strings into account, so any string inputs that will be required will have a concrete value.
numbers - We use ptc solver to solve the constraints to generate number inputs to take the path that we require.
objects - Using the constraints applied to the object up to a certain point, we can
innerHTML - When innerHTML is used to set the content of an element, the string value is parsed into a DOM structure. As explained in the functional spec, the following example expects the resulting html to have a specific structure.

```
/*
* Create a form with the specified content and add an event listener to
* the last child which should be a button.
*
* @param {String} content A string of html to set as the content of the page.
* @param {Function} cb A callback function for when the last child is clicked.
*/
function createForm(content, cb) {
 var form = document.createElement(“form”);

 form.innerHTML = content;

 form.lastChild.addEventListener(“click”, cb, false);

 return form;
}
```

To generate the error we build a DOM structure that would cause an error. Converting the structure back to a HTML string is a simple operation that requires calling innerHTML.

Usage

JSSeek will exist as both a node.js module as well as a command line tool. This will allow it to be included in a web service and also from the command line.

Input

JSSeek will take input as a string representation of some javascript code.

Output

The output will be an AST that is identical to the one created by the parser in the first stage of analysing the code. On any node where an error can occur there will be defined an array of the errors that can occur and the inputs required to cause each error.

```
{
  errors: null | [
       {
          description: string
          arguments: array of arguments
       }
  ]
}
```

Dependencies

Tools

ptsolver -

Node Modules

Esprima - http://esprima.org/ - https://github.com/ariya/esprima
JSDOM **-** https://github.com/tmpvar/jsdom

Bibliography

1. https://developer.mozilla.org/en-US/docs/JavaScript [Accessed 28th November 2012]
2. https://developer.mozilla.org/en-US/docs/Rhino [Accessed December 14th]
3. http://nodejs.org/ [Accessed 28th November 2012]
4. http://phantomjs.org/ [Accessed 28th November 2012]
5. http://www.jshint.com/ [Accessed 28th November 2012]
6. http://www.jslint.com/ [Accessed 28th November 2012]
7. **Saxena, Prateek.** *A Symbolic Execution Framework for JavaScript.* In SP '10 Proceedings of the 2010 IEEE Symposium on Security and Privacy. Pages 513-528, 2010
8. **C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert.** *Rozzle: De-cloaking internet malware***.** In IEEE Symposium on Security and Privacy, May 2012
9. http://jquery.com/ [Accessed 29th November 2012]
10. http://documentcloud.github.com/backbone/ [Accessed 29th November 2012]
11. http://documentcloud.github.com/underscore/ [Accessed 29th November 2012]
12. http://esprima.org/ [Accessed November 27th 2012]
13. https://developer.mozilla.org/en-US/docs/SpiderMonkey/Parser_API [Accessed 11th January 2013]
14. http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts [Accessed 12th January 2013]
15. http://kangax.github.com/es5-compat-table/ [Accessed 12th January 2013]