Sherlock: A Profiling Tool to Find Opportunities for Early Property Initialization

Tarek Auel & David Güsewell

tarek.auel@stud.tu-darmstadt.de, davidnikolay.guesewell@stud.tu-darmstadt.de

Abstract

We present a new dynamic analysis tool for JavaScript, named Sherlock. It uses the dynamic analysis framework Jalangi. During the analysis Sherlock is able to detect opportunities for early property initialization and give a structured feedback. It is able to handle with many complex program structures like if-conditions or while-loops and it support 27 built in JavaScript functions. One main strength of Sherlock is, it keeps the false positives as low as possible. With the proposed optimizations Sherlock can improve the readability of the analyzed code and therefore the maintainability.

1. Introduction

Within the course Program Testing and Analysis we were given a course project with the title „A Profiling Tool to find opportunities for Early Property Initialization“. The goal of the project was to design and develop a dynamic program analysis that finds potential opportunities for early initialization of object properties and array elements (at the time of an object/array creation) that are originally added later during the program execution. The tool should be able to analyze JavaScript files and use the dynamic analysis framework Jalangi (Sen and Gong 2013). Later the developed tool should be tested against Octane (Daishi Kato 2012), a benchmark for node.js The expected benefits from tool were (i) improved code readability, and (ii) improved performance.

In the following section we will firstly introduce the goals of our developed tool in a more detailed way. Section 3 then explain the main techniques behind our tool Sherlock, how it uses Jalangi and the main capabilities of it. The Evaluation of Sherlock is in Section 4. Finally Section 5 briefly summarize our approach.

2. Goal

The goal of the developed profiling tool is to find opportunities for early object property and array element initializations as shown in listing 1. The array `a` could be initialized with `[1]`. In order to do this the analysis has to keep track of the usage of the elements or properties.

```
1 var a = []
2 a.push(1);
```

Listing 1. Example for early array element initialization.

If an initialization is conditioned, e.g. in a branch of an `if` clause or in the body of a loop, it must not be optimized, because the optimization is known to be potentially invalid for many inputs. Besides, an element or property that has been read cannot be optimized afterwards, because this would change the behavior of the program. But if the scope does allow to do an optimization it has to be done. In listing 2 `a` cannot be optimized but `b` can.

```
1 var a = []
2 if (cond) {
3   a[0] = 1;
4   var b = []
5   b[0] = 1;
6 }
```

Listing 2. Example for early array element initialization.

Many default javascript function do exist that manipulate an array, such as `reverse`. If a function can be optimized this should be identified by the profiling tool, too. Listing 3 shows an example of unoptimized code (line $1-3$) and the optimized version (line $5-6$).

```
1 var a = [1, 2, 3]
2 a.push(4);
3 b = a.reverse();

5 var a = [4, 3, 2, 1]
6 b = a;
```

Listing 3. Example for an optimaziation.

Only objects that are initialized by the object notation `{}` are considered. The objects may define already some properties such as `{a: "name"}`. Objects that are created using a constructor are not going to be considered, because optimizations may introduce many false positives. Listing 4 shows a constructor with one parameter. Neither `this.name = 'bmw'` in line 2 nor `var c = new car('bmw');` in line 6 is equal to the original coding. In order to reduce the number of false positives and to limit the scope of the tool, constructors are not taken into account.

```
1 function car(name) {
2     this.name = name;
3     this.knownName = (name !== undefined);
4 }

6 var c = new car();
7 c.name = 'bmw';
```

Listing 4. Example for an optimaziation.

One of the main goals of Sherlock is to keep the number of false positives as low as possible. Given that in some use cases it prefers not optimizing a value instead of running in a potential false positive situation.

3. Sherlock

Sherlock is the developed profiling tool. Overall Sherlock is an implemented plugin for Jalangi which is introduced in the following subsection 3.1. Before the code is analyzed by Sherlock it is instrumented using Esprima (Hidayat 2012), Estraverse (Suzuki

2012b) and Escodegen (Suzuki 2012a) in order to be able to track events that are not exposed by Jalangi such as the end of a branch (Figure 1). How this is done explicitly is explained in subsection 3.8. This additions allows to safely optimize coding which could not have optimized otherwise.

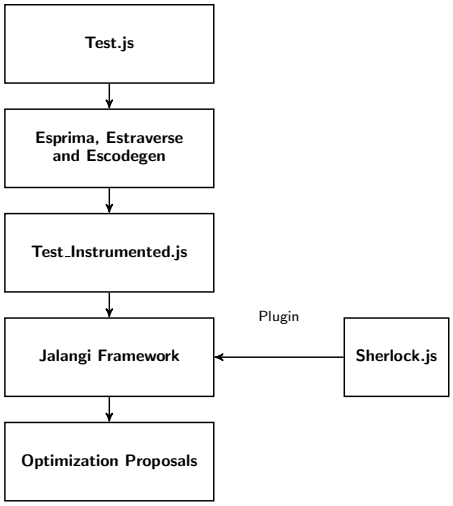

Figure 1. Optimization process using Sherlock with Jalangi.

3.1 Jalangi

Jalangi is a dynamic analysis framework for Javascript. It does allow to implement plugins which implement certain functions. Jalangi calls the functions of the plugin, whenever the corresponding event does occur. The functions that Sherlock uses are explained in the following paragraphs:

- **binary** The `binary` callback is called after the execution of a binary operation, i.e. `<, +, ==, !=, >=`.
- **conditional** The `conditional` callback is called after a condition check before branching, i.e. `if, switch, while, &&, ||`.
- **endExecution** The `endExecution` callback is called, when the execution terminated in node.js.
- **endExpression** The `endExpression` callback is called, when an expression is evaluated and its value is discarded.
- **invokeFunPre** The `invokeFunPre` callback is called, before a function is invoked.
- **functionEnter** The `functionEnter` callback is called, when a function is entered.
- **functionExit** The `functionEnter` callback is called, when the execution of the body of a function finishes.
- **getFieldPre** The `getFieldPre` callback is called, before a value of a property or element is read, i.e. `var x = a.name`.
- **putFieldPre** The `putFieldPre` callback is called, before a value of a property or element is set, i.e. `a.name = 5`.
- **literal** The `literal` callback is called, after the creation of a literal, i.e. `'bmw'`.
- **write** The `write` callback is called, before a variable is writte, i.e. `a = b`.

These are all hooks that Sherlock uses in order to profile the execution of a program. The plugin maintains some variables during the execution:

- **callStack** is an array of `strings` that represent the current call stack. The call stack is cleared, if `endExpression` is called.
- **allRefs** is an array of `References` that the plugin tracks at the moment.

Some other variables are explained later, when their purpose is explained.

3.2 Reference objects

The central data structure is called `Reference`. For every object or array that Sherlock tracks exactly one `Reference` object does exists. One `Reference` object may keep track of multiple references to the object. A UML representation can be found in Figure 2. The properties are explained first. The general purpose functions are explained too. All other functions are explained later, when the analysis is described in more detail.

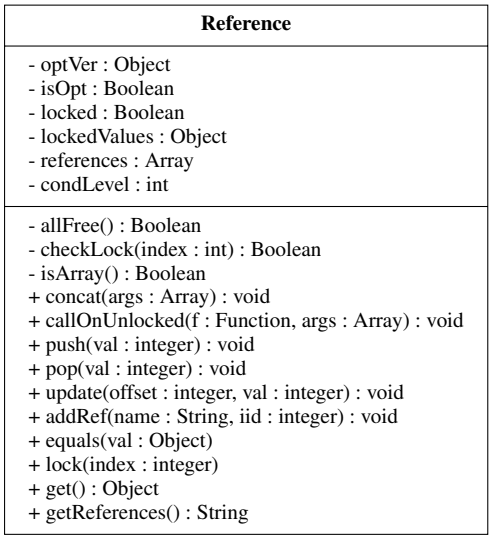

Figure 2. Central data structure to track references.

The following enumeration explains the purpose of all properties of a reference:

- **optVer** is a copy, not a reference, of the object that is represented by this reference. This is not a deep copy. Properties of this object, which are references, refer to the same object as the original object. Primitive values are copied.
- **isOpt** is a boolean flag, which is `false` if the object has not been optimized and `true` if it has been optimized.
- **locked** indicates if the reference is locked. If a reference is locked, it must not be optimized further, e.g. because it has been read already.
- **lockedValues** is an associative array. Each element states if an array index or an object property is locked. If it is locked, it must not be changed anymore. The key of the arrays are the index or property name and the value is `true`. If an entry does not exists it is considered to be unlocked. Typically locked elements / properties do have a reference in this array. The associative array is implemented by using an object.
- **references** is an array of objects. Each object represents a reference to this object. The only property of an object in this array is `name`. Nevertheless, object is chosen as type in order to be flexible to extend this with additional information, e.g. line of code.

- **condLevel** is an integer value that specifies the depth of conditions where this object has been created. This is necessary in order to do the optimizations correctly even if conditionals are nested.

In the following list the general purpose methods are explained:

- **allFree** is a function which checks whether the reference is not locked (`locked === false`) and no element or property is locked. The function is used for example, if a function such as `reverse()` manipulates all elements of an array.
- **checkLock** checks if a specific element or property is locked. If it is locked, it must not be modified. If no index is provided, it checks if the reference is not blocked and if the current condition level allows to modify the array. The conditional lock mechanism is explained later in more detail. It is important to mention that `checkLock` may return `true`, even though `allFree` returns `false`. Listing 5 is an example that highlights the difference.
- **isArray** is a simple check which returns `true` if the reference refers to an Array and `false` if it refers to an object.
- **update** tries to update an array element or an object property. The `offset` parameter specified the element the `val` argument the value of the element. Before the value is written `checkLock (offset)` checks whether an optimization is allowed or not. If yes, it is performed and `isOpt` is set to `true`. If not, the `lockedValues[offset]` is set to `true`, because this potential optimization could not be done. Given that, a value assignment that comes afterwards can not be optimized because it would be overridden by the rejected one.

```
1 {
2   locked: false,
3   lockedValues: {
4     0: true
5   }
6 }
```

Listing 5. Difference of `allFree` and `checkLock`. `allFree` returns `false`, `checkLock` returns `true` for any value including `undefined` except of `0`.

- **addRef** adds a reference to this reference. That does mean that a new identifier refers to the object that is represented by the reference object. The `name` and the `iid` is provided. The former one is stored. The latter one could be used to get additional information such as the line of code.
- **lock** locks the full reference (sets property `locked = true`) if no index is provided. If an index is provided the according value in `lockedValues` will be set to `true`.

3.3 Capturing a new array or object

The creation of an object itself is not captured in Sherlock, but the assignment to a variable , by using the `write` callback. Whenever write is called, Sherlock checks if the object or array that is assigned, is already represented by a reference in `allRefs`. If yes, a new reference with the variable name is added to the references. If not, a new `Reference` object is created and append to `allRefs`.

Typically an object or array and all its properties or elements are unlocked. Listing 6 shows an example where the reference is locked immediately. The reason is that `someFunction()` may have arbitrary side effect and must not be optimized. Unfortunately, it is almost impossible to determine the index of the element that is assigned by the function call. Hence, the reference is locked in order to decrease the probability of false positives.

```
1 var a = [0, someFunction()];
2 a[0] = 1;
3 a[1] = 2;
```

Listing 6. Example for an array that is immediately locked after creation. Neither index `0` nor index `1` is optimized.

3.4 Optimize array elements or object properties

Listing 7 shows a typical example for a potential early property and array element initialization. Line 2 and line 5 could be removed and the assignment added in line 1 and line 4.

```
1 var a = [];
2 a[0] = 1;

4 var b = {};
5 b.a = 5;
```

Listing 7. Example for a typical potential early property and array element initialization

Sherlock captures this using the `putFieldPre` callback. The new value and the reference to the object is given. Using the reference Sherlock can get the reference object and store the value in a temporary storage. At the end of a statement it is checked if a `functionExit` is part of the call stack. If yes, the element or property is not modified but locked, because a function call was potentially involved by creating the value. Because we cannot do any statement about side effect of the function call, it cannot be optimized. If not, the property or element is modified and optimized. The `update` function checks still if the optimization is allowed. The variable `lastPut` is used to store the necessary information of `putFieldPre` until the put is evaluated in `endExpression`.

3.5 Lock read values

Whenever an object property or an array element is read, it cannot be optimized further. The callback `getFieldPre` allows to easily detect when a property or element is read. Because Javascript uses references, the reference can be used to identify the reference and to call `lock(offset)` with the offset that was used. If the `length` of an array is read, it might be necessary to lock the reference. Listing 3.5 shows two different examples. The read of `a.length` is okay and equal to `a.push`. Hence, the optimized version of `a` would be `[0, 1]`. But the `b` must not be optimized to `[1, 2, 3, 4]`, because this would change the semantics. Sherlock can deal with that by storing the `getFieldPre` call in a temporary storage. At the end of the execution it is analyzed if `length` was used as synonym for `push`. `c` and `d` show two corner cases that are not supported. It turns out that it is very hard to distinguish `c` and `d` using Jalangi. Therefore, Sherlock does only support `length` as synonym for `push`, if no further `binary` operation is involved.

```
1  var a = [];
2  a[a.length] = 0;
3  a[1] = 1;

5  var b = [1, 2, 3];
6  console.log(b.length);
7  b[3] = 4;

9  var c = [];
10 c[c.length - 1] = 0;

12 var d = [];
13 d[0] = d.length - 1;
```

3.6 Handle transformation functions

So far it is explained how Sherlock handles property / array element updates and reads. But Javascript does have many built in functions

that either manipulate the array or that do provide other capabilities. Listing 8 shows a function call that can be optimized. Sherlock separates the built in functions in four groups:

```
1 var a = [1, 2, 3, 4];
2 a.revere();
```

Listing 8. Example for handle transformation functions.

- **Ignore** Methods that belong to the group do not change the state or do effect the reference in any way. The only method that belongs to this group is `Object.prototype.isPrototypeOf`.
- **Lock reference** Methods that belong to this group do lock the reference, because an optimization that optimizes a value that is written after the call would impact the return value. The following methods belong to this group:
  - `Object.prototype.toString`
  - `Object.prototype.toLocaleString`
  - `Object.prototype.valueOf`
  - `Object.prototype.hasOwnProperty`
  - `Array.prototype.toString`
  - `Array.prototype.toLocaleString`
  - `Array.prototype.indexOf`
  - `Array.prototype.lastIndexOf`
  - `Array.prototype.every`
  - `Array.prototype.some`
  - `Array.prototype.forEach`
  - `Array.prototype.map`
  - `Array.prototype.reduce`
  - `Array.prototype.reduceRight`
  - `Array.prototype.join`
  - `Array.prototype.filter`
  - `Array.prototype.slice`
- **Call on unlocked** Methods that belong to this group can only be called if `allFree()` return `true`. That does mean that neither the reference nor any value is locked. These methods do modify all values of the object. The `reverse` function in listing 8 is an example for this group. The function `callOnUnlocked` (see figure 2) gets the function and the arguments for the function as arguments. If `allFree()` returns `true`, it applies the operation to the object. If not, it simply sets `locked` to `true`, because no further optimization can be applied.
  - `Array.prototype.reverse`
  - `Array.prototype.shift`
  - `Array.prototype.sort`
  - `Array.prototype.splice`
  - `Array.prototype.unshift`
- **Other** All other methods belong to this group, because they cannot be added to one of the previously mentioned groups. The reference object does have a corresponding method for all these methods (see figure 2):
  - `Array.prototype.concat` This method can be applied, if the reference is not locked. It does not matter whether one of the values is locked. Besides, this method can have an arbitrary number of arguments.
  - `Array.prototype.push` The `push` method requires that the reference is not locked and that the element with the index `array.length` is not locked.
  - `Array.prototype.pop` The `pop` method requires that the reference is not locked and that the element with the index `array.length - 1` is not locked.
  - `Object.prototype.propertyIsEnumerable` If this method is called the corresponding value is locked and must not be optimized further.

3.7 Dealing with functions

In all optimizations that Sherlock may propose functions do require a special investigation. Whenever a function is part of a statement Sherlock cannot be sure, whether the function has some side effects or not. In Figure 6 we already explained, why we cannot optimize an array further, which has at least one element that is initialized with the return value of a function. But this is not the only case where functions must not be optimized. In Figure 9 there are two additional examples: in line 3 the return value of a function is assigned to an element. One must not optimize it, by initializing the array with the return value. But it would be allowed to move the function to the initialization. Sherlock does not recommend this, because the function might use variables which are initialized after the array initialization and it reduces the readability if the array initialization contains function calls. in line 7 the array index is given by the return value of a function. This must not be optimized, because the one cannot assume that the return value is always the same. Besides, side effect could be lost. Hence, both cases must not be optimized and Sherlock does simply ignore them and knows that the elements may not be optimized further.

```
1 var a = [];
2 a.push(1);
3 a[1] = (function(){return 0;})();

5 var b = [];
6 b.push(1);
7 b[(function(){return 1;})()] = 1;
```

Listing 9. Example how Sherlock handles functions.

3.8 Dealing with conditioned updates

Sherlock can not simply optimize any update that is applied to a property or element, even if the property or element has not been read yet. Listing 10 shows a simple example that shows which updates are allowed and which are not. `a` can safely optimized to `[5, 2, 3]` and `b` to `[1, 3, 3]`. The statement in line 4 must not be optimized, because Sherlock does not know, if `condition` will always be `true`. The same applies for the statement in line 6. Even tough line 9 is always executed if line 3 is executed, it must not be optimized. The reason is that if `otherCondition` is `true` `b` would be 7 at the end of the execution instead of 5. In this particular scenario `b[0]` could be optimized to 5 and line 7 and 10 could be removed. But because Sherlock does not keep track of property writes that are conditioned but are not read before they are updated unconditioned, it simply locks the value in line 7. This follows the goal to minimize the number of false positives and prefer being too restrictive.

```
1 var a = [1, 2, 3];

3 if (condition) {
4   var b = [1, 2, 3];
5   a[1] = 7;
6   if (otherCondition) {
7     b[0] = 7;
8   }
```

```
9     b[1] = 3;
10    b[0] = 5;
11    var c = a[2];
12  }

14  a[0] = 5;
15  a[2] = 0;
```

Listing 10. Example which optimizations can be done dealing with conditioned updates

Jalangi does have a callback for conditionals which is called whenever a conditional check is performance. But Jalangi has no callback to identify the end of the branching. In a first version of Sherlock assumed that a branching finished at the end of the current function. This is very restrictive and limits the capabilities, especially because arguments of functions are often checked by a couple of `if` statements before the actual logic of a function is implemented. Sherlock had no chance to optimize arrays or objects in this use cases. But the problem is partially solved.

Before code is executed it is instrumented with a simple literal `'da0b52b0ab43721cda3399320ca940a5a0e571ee'`. This literal is a signal for Sherlock that a conditioned branch finished. An example for instrumented code can be seen in figure 11. Jalangi calls the callback for conditionals multiple times, if the condition exists out of multiple binary equations. Because of that the literal appears twice (line $7-8$). A condition is checked $n+1$ times for a loop which has n iterations. This is solved by adding the literal at the end of the body loop and immediately after the end of a loop.

```
1   // ...
2   while (condition) {
3       // ...
4       var x = []
5       if ((a == b)  (b == c)) {
6           // ...
7       }
8       'da0b52...ee'; //  of if
9       'da0b52...ee'; // if statement
10      // ...
11      for (i = 0; i < 5; i++) {
12          // ...
13          'da0b52...ee'; // end of iteration
14      }
15      'da0b52...ee'; // last check of for
16      // ...
17      'da0b52...ee'; // end of iteration
18  }
19  'da0b52...ee'; // last check of while
20  x[0] = 1;
21  // ...
```

Listing 11. Example how the code is intrumented by Sherlock in order to deal with conditions.

Each reference has a `condLevel` property (see figure 2). Besides a global counter does exist which contains the current level of conditions. In listing 11 the level would be 0 in line 1, 1 in line 4, 3 in line 6, 1 in line 10 and so on and so forth. `condLevel` is set to the conditional level if a reference is created. Each optimization has to be on the same level as `condLevel`. If the level is higher, it must not be applied. If the global conditional level is lower than `condLevel`, the reference is locked. Javascript hoists the definition of variables to the scope of the current function. A variable cannot be created in the scope of an `if`. The variable `x` can be accessed in line 20 in listing 11. By locking the `condLevel` it is prevented that a false optimization is done. If the reference would not be locked, Sherlock would think that it can optimize `x` in listing 12.

```
1   if (condition) {
2     // condition level = 1
3     var x = [];
4   }
5   // x is locked

7   if (otherCondition) {
8     // condition level = 1
9     x[0] = 5;
10  }
```

Listing 12. Example showing the use of condLevel property.

4. Evaluation

In order to evaluate the results, it was a given requirement to check, whether Sherlock can find anything that could be optimized in the Octane benchmark. The results of that are presented first, afterwards a general evaluation of the capabilities of Sherlock is given.

4.1 Performance improvements by Sherlock

Sherlocks helps finding opportunities for early property or element initialization. Most of the time this improves the readability. Because some statements are discarded, there might be a performance improvement, too.

The following benchmarks has been executed on a MacBook Pro Late 2013 Intel(R) Core(TM) i7-4750HQ CPU @ 2.00GHz. v0.12.9 is the used nodejs version. `gdate` is the GNU coreutils implementation of the UNIX `date` function. The command that is used to measure the performance can be found in Listing 13. We decided to add the loop outside of nodejs in order to not run into just in time compiler optimizations.

```
1   gdate +%s%3N; for i in {0..1000}; do node
        optimized.js 2>&1 >/dev/null; done; gdate
        +%s%3N
```

Listing 13. Performance testing commands

Listing 14 shows the unoptimized code. It basically exists of 100 push calls. In order to avoid that JavaScript removes them, the array is printed to `stdout`. The optimized coding in Listing 15 initializes the array and prints it immediately. Table 1 shows the measured runtimes. The first two lines show the runtime in milliseconds. The second and third line shows the runtime difference. Surprisingly, the unoptimized version is even faster for both number of iterations. The fifth and sixth line show the time that was needed per element. The last two lines show the runtime of the unoptimized version in relation to the optimized one.

```
1   var a = [];
2   a.push(1);
3   a.push(2);
4   a.push(3);
    ...
101 a.push(100);
102 console.log(a);
```

Listing 14. Unoptimized test coding

```
1   var a = [
2   1,
3   2,
4   3,
    ...
101 100];
102 console.log(a);
```

Listing 15. Optimized test coding

The performance test shows that the optimized code might be even slightly slower. Because of incompatibility issues with Jalangi,

# Iterations	Optimized Coding	Unoptimized Coding
1,000	55,086 ms	54,831 ms
10,000	554,431 ms	553,333 ms
1,000	Δ 0 ms	Δ −255 ms
10,000	Δ 0 ms	Δ −1,098 ms
1,000	55.09 ms/element	54.83 ms/element
10,000	55.44 ms/element	55.33 ms/element
1,000	100 % runtime	99.54 % runtime
10,000	100 % runtime	99.80 % runtime

Table 1. Measured runtimes in ms including the runtime difference and the runtime per element.

Sherlock is forced to use a very old version of nodejs. In newer versions of nodejs the optimized code could be faster.

4.2 Octance

Unfortunately Sherlock is not capable to find any optimization in the Octane benchmark. Because there was no obvious reason for that a more detailed analysis was required.

We looked intensively at the `Richards` benchmark and the `Splay`. If one analyzes the coding of the benchmarks he can identify quickly why Sherlock could not find any optimization. First of all these benchmark use almost no plain object initialization. Sherlocks does not optimize constructors, because of the reasons mentioned earlier. But plain objects seems to be of almost no interest of the benchmark. Arrays are used sometimes in the benchmarks. The reason that Sherlock could not find an array optimization is that there are no. If arrays are used they are often initialized with values. If they are initialized as empty arrays, items are often added in loops, conditioned branches, function calls. An optimization can not be applied safely, if it is conditioned. The last thing that one notices while analyzing the benchmark, they do not utilize the default object or array methods. One of the focuses of Sherlock is optimizing built in JavaScript functions. Because the benchmarks do not use them, they could not be found.

In order to check, whether Sherlock is capable to find an optimization in the coding, we instrumented the coding with plain objects and arrays that could be optimized. Listing 16 shows the beginning of the function `runRichards` which we instrumented with a couple of statements. Sherlocks correctly determines that `a` can be optimized to `[1, 6]` and `b` to `[5, 2, 3]`. This shows that Sherlock can optimize code within benchmarks like Octane.

```
1  // ...
2  function runRichards() {
3
4    var abc = [];
5    abc.push(1);
6
7    if (true) {
8      // do nothing
9      var b = [1, 2, 3];
10     while(true) {
11       b[1] = 3;
12       break;
13     }
14     b[1] = 3;
15     b[0] = 5;
16   }
17
18   abc.push(6);
19
20   var scheduler = new Scheduler();
21   // ...
```

Listing 16. Richards

4.3 Sherlocks capabilities

Even though Sherlock can cope already with many complex program structures, there are still some patterns known, where Sherlock is either too restrictive or doesn't get them right. On of these code patterns is a `do - while` loop. The problem can be seen in Listing 17 (line 1 – 5). The instrumented literal in Line 2 which tells Sherlock the end of a branching has to be only executed in the second execution of the loop or later. The literal in Line 5 is used for the last check of `cond` which leaves the loop. Even though different patterns exist to mitigate this problem, Sherlock doesn't do this, yet. The first proposal would be to copy all statements and execute them before a `while` loop (line 7 – 12). The second proposal would be to add a counter which masks the literal in line 2 so that it is not executed in the first iteration (line 14 – 21). Both solutions do modify the coding more heavily. The counter must have a name that was not used before. This becomes even more tricky if the loops are nested. Is it guaranteed that the semantics may never change, if we copy the statements (1st approach)? Someone could do something that depends on the line number of the coding. If Sherlock copies the coding, there are two different line numbers for the statement. Because of all these difficulties, we decided to ignore this in this stage of development of Sherlock and decide later which Solution we do prefer.

```
1  do {
2    'da0b52...ee'; // but only if not 1st loop
         iteration
3    // some statements
4  } while(cond);
5  'da0b52...ee';
6
7  // some statements
8  while(cond) {
9    // some statements
10   'da0b52...ee';
11 }
12 'da0b52...ee';
13
14 var i = 0;
15 do {
16   if (i != 0) 'da0b52...ee';
17   'da0b52...ee'; // literal for the introduced
         if
18   // some statements
19   i++;
20 } while(cond);
21 'da0b52...ee';
```

Listing 17. Example for do-while loop and how Sherlock can handle them in future iterations.

So far Sherlock returns only an optimized version for each object or array that could be optimized. Because the preprocessing changes the lines of code it does not make sense to return the lines of code that Jalangi proposes. A mapping to the original lines of code could improve the usability of the results of Sherlock.

All in all Sherlock supports 27 built in JavaScript functions. Due to the fact, that Sherlock tries to report as few false positives as possible it is sometime more restrictive than it has to be. With a deeper tracking of the array elements, it would be possible to allow further optimizations after a `Array.prototype.slice`

call. But the real world scenarios where this applies are rare. Sherlock can deal with conditioned writes and does know, whether an optimization is safe or not. Even nested structures or loops can be handled. So that, Sherlock does help to improve the readability and therefore the maintainability of coding.

5. Summary

The goal of the course project was to develop a new dynamic analysis tool for finding opportunities for early property initialization in order to improve the performance and the readability of JavaScript code. While there is still a long way to go to find all opportunities, our results show that Sherlock is able to cover a broad range of cases which can be optimized. We also give some ideas for future iterations of Sherlock to handle do - while loops. Our tool is able to automatically generate proposals for early property initialization and keep the number of false positive as low as possible in the same time. This is really important for the usability of such a dynamic analysis tool. So far Sherlock report only the optimized version of an object and cannot tell the programmer which coding lines could be removed.

Our results show that the goal to improve the readability of the code can be achieved by Sherlock, but there is no improvement in the performance. This may be due the fact that we used a very old version of nodejs because Jalangi has some incompability issues with newer versions.

Acknowledgement

We thank M. Selakovic for the mentoring during the project, K. Sen and L. Gong for Jalangi, A. Hidayat for Esprima and Y.Suzuki for Estraverese and Escodegen.

References

E. R. Daishi Kato. Octane. `https://github.com/dai-shi/benchmark-octane`, 2012. [Online; accessed 09-February-2016].

A. Hidayat. Esprima. `http://esprima.org`, 2012. [Online; accessed 04-February-2016].

K. Sen and L. Gong. Jalangi - A Dynamic Analysis Framework for JavaScript. `http://srl.cs.berkeley.edu/ ksen/papers/jalangi-tool.pdf`, 2013. [Online; accessed 04-February-2016].

Y. Suzuki. Escodegen. `https://github.com/estools/escodegen`, 2012a. [Online; accessed 04-February-2016].

Y. Suzuki. Estraverse. `https://github.com/estools/estraverse`, 2012b. [Online; accessed 04-February-2016].