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

Tarek Auel & David Güsewell

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1. Introduction

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.

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

Figure 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.

```
var a = []
if (cond) {
  a[0] = 1;
  var b = []
  b[0] = 1;
}
```

Figure 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 $5-6$).

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

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

Figure 3. Example for an optimaziation.

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 }
5
6 var c = new car();
7 c.name = 'bmw';
```

Figure 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

3.1 Introduction

Sherlock is the developed profiling tool.

3.2 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`.
- **getFieldPre** 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.

3.3 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 5. 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.

Reference
- optVer : Object - isOpt : Boolean - locked : Boolean - lockedValues : Object - references : Array - condLevel : int
- allFree() : Boolean - checkLock(index : int) : Boolean - isArray() : Boolean + concat(args : Array) : void + callOnUnlocked(f : Function, args : Array) : void + push(val : integer) : void + pop(val : integer) : void + update(offset : integer, val : integer) : void + addRef(name : String, iid : integer) : void + equals(val : Object) + lock(index : integer) + get() : Object + getReferences() : String

Figure 5. 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 6 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.

```
{
  locked: false,
  lockedValues: {
    0: true
  }
}
```

Figure 6. 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`.

3.4 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 7 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.

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

Figure 7. Example for an array that is immediately locked after creation. Neither index `0` nor index `1` is optimized.

3.5 Optimize array elements or object properties

Listing 8 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;
3
4 var b = {};
5 b.a = 5;
```

Figure 8. 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`.

3.6 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 9 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.

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

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

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

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

Figure 9.

3.7 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 10 shows a function call that can be optimized. Sherlock separates the built in functions in four groups:

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

Figure 10.

- **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 10 is an example for this group. The function `callOnUnlocked` (see figure 5) 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 5):

  - `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.

3.8 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 11 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.

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

```
1  var a = [1, 2, 3];
2
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 }
13
14 a[0] = 5;
15 a[2] = 0;
```

Figure 11.

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 12. 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 // ...
```

Figure 12.

Each reference has a `condLevel` property (see figure 5). Besides a global counter does exist which contains the current level of conditions. In listing 12 the level would be 0 in line 1, 1 in line 4, 3 ind line 6, 1 in line 10 and so 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 12. 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 13.

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

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

Figure 13.

4. Evaluation

References