INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
LFE PROGRAMMIERUNG UND SOFTWARETECHNIK

Master Thesis

SMT-Based Verification of ECMAScript Programs in CPAchecker

Michael Maier

INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
LFE PROGRAMMIERUNG UND SOFTWARETECHNIK

Master Thesis

SMT-Based Verification of ECMAScript Programs in CPAchecker

Autor: Michael Maier
Matrikelnummer: 10374559
Aufgabensteller: Prof. Dr. Dirk Beyer
Betreuer: Dr. Philipp Wendler
Abgabetermin: 07.05.2019

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

Munich, May 7th 2019

.....

(Signature of the author)

Abstract

Software model checking is a successful technique for automated program verification. The configurable Software-Verification Platform CPAchecker supports several approaches based on solving first-order-logic formulas over predicates using SMT solvers. The core of CPAchecker including all algorithms is not specific to a programming language. CPAchecker supports the static languages C and Java, but no SMT-based approach is supported for Java yet.

In this paper, we extend CPAchecker to a restricted subset of ECMAScript 5.1 by adding a respective parser frontend and an operator that is responsible for encoding the semantics of program operations into SMT formulas. JavaScript is one of the most widespread dynamic languages and conforms to the ECMAScript specification. Its dynamic nature and complex semantics make it a difficult target for logic-based verification. In defining the formula encoding (operator), we deal with dynamic types, implicit type conversion, internal method calls, function objects (higher order functions), closures (scope chain), extensible objects, dynamic property access, and prototype inheritance. The functional correctness of the implementation of the formula encoding is evaluated based on the test programs of the official ECMAScript Conformance Test Suite using bounded model checking and k-induction. Finally, we outline what needs to be done to be fully ECMAScript 5.1 compliant.

Contents

1 Introduction

JavaScript is one of the most widespread dynamic languages. It is the main language for Web applications, used on the server-side via Node.js, in desktop applications using Electron, and in mobile applications using PhoneGap or Ionic. It is even run on small embedded devices with limited memory using low.js[1]. According to W3Techs it is used by 95.1% of websites[2]. Further, it is the most active language in GitHub[3]. The Stack Overflow Developer Survey 2019[4] showed that JavaScript is the most commonly used programming language (for the seventh year in a row).

The understanding and development of correct JavaScript code is notoriously difficult. This is due to the dynamic nature and complex semantics of the language. There are less specialized static analysis tools for JavaScript than for statically typed languages such as C and Java. The transfer of analysis techniques to the domain of JavaScript (and dynamic languages in general) is known to be a challenging task. Automated verification of safety properties can help to find bugs and security vulnerabilities in JavaScript programs.

JavaScript is an interpreted language that is executed by a JavaScript engine. JavaScript engines are commonly found in web browsers, including Chakra in Edge, SpiderMonkey in Firefox, and V8 in Chrome. Each engine implements a different dialect of the JavaScript language. The JavaScript language itself is not standardized, but Ecma International[5] specifies the language ECMAScript in ECMA-262[6] that acts as standard for JavaScript. There exist different versions of ECMAScript. Most engines implement JavaScript conforming to ECMAScript 5.1[7] as described in section conformance [ES5, Sec. 2]. Newer versions are largely backwards compatible to ECMAScript 5.1.

CPAchecker is an award winning configurable software-verification platform for C (and Java). We extend it to a restricted subset of ECMAScript 5.1 to benefit from the various SMT-based analysis approaches provided by this platform. The dynamic nature and complex semantics of ECMAScript make it a difficult target for logic-based verification. We first look at the challenges and related work in Chapter 2. Then we explain the SMT-based verification approach of CPAchecker and outline what is needed to extend it to another language in Chapter 3. This leads to Chapter 4 that describes how a ECMAScript program is represented and to Chapter 5 that describes how the SMT-formula encoding is done. Chapter 6 addresses the implementation of the CPAchecker extension. The evaluation of the functional correctness of the implementation is subject of Chapter 7. Finally, we summarize the work in Chapter 8 and outline future work in Chapter 9, especially what has to be done to be fully ECMAScript 5.1 compliant.

[1] https://www.lowjs.org/
[2] https://w3techs.com/technologies/details/cp-javascript/all/all
[3] http://githut.info
[4] https://insights.stackoverflow.com/survey/2019/#technology-_-programming-scripting-and-markup-languages
[5] https://www.ecma-international.org/
[6] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[7] http://kangax.github.io/compat-table/es5/

2 Related Work

Santos et al.[27] identified several challenges that need to be addressed for tractable ECMAScript verification to be possible. We have to reason robustly and abstractly about the ECMAScript internal functions that are used in the ECMAScript standard to (operationally) describe the semantics of statements and expressions. This includes object property management (e.g. creation (`DefineOwnProperty`), lookup (`GetValue`), modification (`PutValue`) and deletion (`Delete`)) and type conversions (e.g. `ToString` and `ToNumber`). Further, we have to reason about extensible objects (properties can be added and removed from an object after its creation), dynamic property access (we cannot guarantee statically which property of the object will be accessed), property descriptors (describe the ways in which a property can be accessed or modified), and property traversal (for example using `for - in` or `Object.keys`). Thereby, we have to reason about prototype chains of arbitrary complexity. Besides, ECMAScript stores functions as objects in the heap. They can be passed to other functions as arguments and may be called dynamically. Hence, we have to reason about higher-order functions of arbitrary complexity. Scope is also tied to function objects. Variables and (nested) functions declared in the function body are created upon the invocation of a function. Thus, we have to reason about scope chains and function closures of arbitrary complexity. We will tackle most of these challenges later in Chapter 5 and discuss in Chapter 8 to what extent we were able to solve these challenges. We then give an outlook in Chapter 9 on how our approach can be expanded in future work.

But first we take a look at related work. A number of papers deal with type analysis of JavaScript [1, 19, 31], whereby some papers [6, 13, 24] are built on the (unsound) type system of TypeScript[1], which is a typed superset of JavaScript (with optional types). Other literature is focused on control flow analysis [14], pointer analysis [18, 30], and abstract interpretation [2, 19, 20, 22]. On the contrary, only a handful of works address logic-based verification of JavaScript or ECMAScript.

$\mathbb{K}$ [25] is a rewrite-based executable semantic framework in which programming languages, type systems and formal analysis tools can be defined using configurations, computations and rules. KJS [23] provides a $\mathbb{K}$-interpretation of the core language and part of the built-in libraries of ECMAScript 5.1. It has been tested against an old version of the ECMAScript language conformance test suite Test262[2] that back then (April 28th 2015) consisted of 11578 test programs. They passed all 2782 tests for the core language. As of May 2019, Test262 consisted of nearly 31000 individual test files. Tests for later specified ECMAScript versions have been added, but also tests for older ECMAScript versions (inluding 5.1) have been added and revised in the meantime.

JaVerT [15] is a semi-automatic JavaScript verification toolchain. It uses separation logic to verify functional correctness properties of JavaScript programs annotated with pre- and post-conditions, loop invariants, and instructions for folding and unfolding user-defined predicates. Santos et al.[28] recently published a new version JaVerT 2.0 that supports automatic compositional testing based on bi-abduction. It is also tested against Test262 (May 30th 2016). They passed 8797 tests that target ECMAScript 5.1 strict code (excluding other files with non-strict code, unimplemented features, etc.).

[1] `https://www.typescriptlang.org/`
[2] `https://github.com/tc39/test262`

As we have seen, Test262 is a common source for test programs to test the coverage of an approach. We will also test our approach against the latest version[3] of Test262 (May 3rd 2019) in the end (see Chapter 7).

[3] `https://github.com/tc39/test262/tree/d47749e84daeea28b6fa7cefd69e7f2836dbbf37`

3 SMT-Based Verification In CPAchecker

CPAchecker[1] is a configurable software-verification platform. Its goal is the automated verification of safety properties of sequential programs. Therefore, it provides different configurable program analyses [32]. Several approaches are based on solving first-order-logic formulas over predicates using SMT solvers. CPAchecker supports the static languages C and Java[2]. However, its core (including all algorithms) is not specific to a programming language. CPAchecker can be extended to other imperative and related programming languages by adding a respective parser frontend and replacing the operator that is responsible for encoding the semantics of program operations into SMT formulas.

The parser frontend creates a *control-flow automaton* (CFA) from the source code that represents the program[3]. The CFA consists of a set of program locations, whereas an initial program location represents the program entry point. Edges connect program locations. Program locations without outgoing edges represent the end of the program. Edges are labeled with an operation that is executed when the control flows along the edge. The semantics of an operation *op* are defined by the *strongest post operator* $\mathrm{SP}_{op}(\cdot)$. A program path is a legal sequence of consecutive edges that starts at the initial program location. The semantics of a program path is defined by the iterative application of the strongest post operator for each operation of the path with $\top$ passed as initial formula. The program path is feasible if the resulting formula is satisfiable. Otherwise, it is infeasible. A location is reachable if a feasible program path to it exists. CPAchecker uses reachability analysis to show that no defined error location is reachable or to find a feasible error path to the respective error location.

In this paper, we extend CPAchecker to a restricted subset of ECMAScript 5.1. Therefore, we add a respective parser frontend that creates a program representation (CFA) as described in Chapter 4. Chapter 5 describes the formula encoding of the ECMAScript operations and the language itself.

[1] `https://cpachecker.sosy-lab.org/`
[2] No SMT-based approach is supported for Java yet.
[3] The program analyses of CPAchecker configure the reachability analysis based on the CFA.

4 Program Representation

We assume that an ECMAScript program meets the assumptions of Section 4.1. It is represented by a set of control-flow automata (CFAs) as described in Section 4.2. The program is simplified syntactically according to the rules described in Section 4.3 by preprocessing during CFA creation.

4.1 Assumptions

We assume that a ECMAScript program is a single file. If a program consists of multiple files, then we treat them as if they were concatenated (in the given order) to a single file that we further call program. We further assume that the program is syntactically correct strict mode code [ES5, Sec. 10.1.1] that only contains ECMAScript 5.1 features[1] except the following features that are not covered yet:

- The program must not contain recursive function calls.
- The program must not contain for-in statements [ES5, Sec. 12.6.4].
- The program must not contain the deprecated[3] with statement [ES5, Sec. 12.10].
- Exceptions are not supported in general, including implicitly thrown runtime errors (for example `TypeError`) and explicitly thrown exceptions using the throw statement [ES5, Sec. 12.13]. That also means that exception handling using try statement [ES5, Sec. 12.14] is not supported.
- The program must not contain debugger statements [ES5, Sec. 12.15]. They are supposed to be used in development only.
- The standard built-in ECMAScript objects [ES5, Sec. 15] are not supported yet. This also includes the global object [ES5, Sec. 15.1]. Global variables are supported, but not set on the global object.
- The program must not contain regular expression literals [ES5, Sec. 7.8.5] since the built-in regular expression objects [ES5, Sec. 15.10] (`RegExp`) are not supported. The string encoding (see Section 5.2.5) we use is not precise enough to support operations of regular expressions in a meaningful way. It would require further elaboration.
- Property attributes [ES5, Sec. 8.6.1] are not fully supported. It is assumed that all properties are named data properties that are writable and configurable. Named accessor properties are not supported. That also means that the production of PropertyAssignment [ES5, Sec. 11.1.5] using `get` and `set` to define named accessor properties is not supported.

[1] Newer versions are largely backwards compatible and many new features are transpilable using Babel[2] or another tool to get ECMAScript 5.1 code that can be analyzed.

[3] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with`

- The ECMAScript specification uses various internal properties, which are not part of the ECMAScript language, to define the semantics of object values. Not all of these semantics are covered in this work, because certain features like exceptions are not supported yet. Implicit function calls from internal methods[4] are not supported either, since they are not represented in the CFA yet.
- The arguments object [ES5, Sec. 10.6] is not covered due to lack of time.
- Relational operators `<`, `>`, `<=`, and `>=` do not compare string values. The presented formula encoding of strings (see Section 5.2.5) is not precise enough to handle this case adequately.

The time of this research was not enough to cover all ECMAScript 5.1 features, but we outline in Chapter 9 how some of these unsupported features might be addressed in future work.

4.2 Control-Flow Automata

A program is represented by a set of *control-flow automata* (CFAs), one for each function (declaration body) of the program and one for (the statements of) the global code. A control-flow automaton (CFA) is a directed graph with control-flow locations (program-counter value) as nodes and program operations attached to the edges. Each CFA has an entry node. In addition, every CFA of a function declaration has one function exit node. The entry node of the CFA of the global code represents the entry point of the program execution. A node without outgoing edges models the end of a program run. CFAs might be connected via edges for function calls (see Section 4.3.1.12).

4.2.1 Operations

The CFAs contain only a subset of ECMAScript statements and expressions plus a special binary operator `declaredBy` (see Section 4.2.2.1). Other statements and expressions are transformed to this subset as described in Section 4.3. Therefrom, operations attached to the edges can have exactly one of the following forms:

- an assumption [`p`],
- a variable declaration `var x` or `var x = e`,
- a function declaration `function func(args*) { ... }`,
- an assignment `lhs = e`,
- a delete operation `delete o.propName` or `delete o[e]`,
- a function call `func(e*)`,
- a constructor call `new func(e*)`

Here, `p` is a predicate, `x` is a program variable identifier, `e` and `o` are expressions, `e*` is an arbitrary amount (list) of expressions (e.g. `e0, e1, e2`), `propName` is a property name (identifier), `args*` is an arbitrary amount (list) of program parameter declarations (identifiers), `func` is a program function (declaration) identifier, `func(e*)` is a call to the function `func` with an arbitrary amount of expressions `e*` passed as parameters and the left hand side `lhs` is either a property-access operator expression or an identifier of a variable or parameter (declaration).

[4] For example, `valueOf` might be called by the internal method *[[DefaultValue]]* [ES5, Sec. 8.12.8].

Note

Function call operations also have an expression that is passed along as `this` argument even though it is not shown in the code `func(e*)`. Instead it is explicitly mentioned in the transformation of function calls (see Section 4.3.1.12).

Note

Function and constructor call operations also may have an expression that is passed along as optional *function object argument* even though it is not shown in the code `func(e*)` or `new func(e*)`. The special function object argument is only passed in the delegation of unknown function calls (see Section 4.3.1.12) and unknown constructor calls (see Section 4.3.1.13). Since it is not shown in the code, it is explicitly mentioned in the transformation description.

4.2.2 Expressions

Expressions and predicates (in operations) refer only to program variable, parameter, or function (declaration) identifiers, the this-keyword and literals (boolean, number, null, string, object, array). Furthermore, they contain only the following operators:

- binary operators `&`, `/`, `==`, `===`, `>`, `>=`, `in`, `instanceof`, `<<`, `<`, `<=`, `-`, `!==`, `!=`, `|`, `+`, `%`, `>>`, `>>>`, `*`, `^`
- unary operators `+`, `-`, `~`, `!`, `typeof`, `void`
- the property-access operators `o.f` and `o[p]` for objects with `o` and `p` being expressions and `f` being an identifier name (string)

Besides, a predicate also might be a `declaredBy` expression (see Section 4.2.2.1).

4.2.2.1 declaredBy Operator

`declaredBy` is a special operator (not part of regular ECMAScript) that is only used in the CFA of the functions that resolve dynamic function calls (see Section 4.3.1.12). It takes two operands `id` and `functionDeclaration` and is displayed in infix notation `id declaredBy functionDeclaration` in the CFA edges. `id` is a variable or parameter identifier. `functionDeclaration` is a function (declaration) identifier. The operator checks if the function object stored in `id` has been declared by the function declaration of `functionDeclaration`.

4.2.3 Identifiers

There exist identifiers for variables, parameters and functions. Each has a globally unique name in the CFA (achieved by renaming identifiers during preprocessing) and refers to a declaration of its kind (declaration of variable, parameter, or function). An exception are *global identifiers* that have not been declared by the program (no variable- or function-declaration operation). These represent global variables (including members of *the global object* [ES5, Sec. 15.1]). Global identifiers are not renamed, since they are already globally unique.

4.2.4 Declarations

Identifiers refer to a declaration of a variable, parameter, or function. Variable declarations may be declared globally or in a function (local variable). Function declarations might be nested in another function (declaration). Local variable, parameter, and nested function declarations have a reference to the function declaration they are declared in.

4.3 Preprocessing

The main goal of the preprocessing is that expressions are free of side effects and only the minimal set of necessary operations stated in Section 4.2 is used. Therefore, each operation that is not part of the minimal set is transformed into an equivalent sequence of operations that are all part of the minimal set. The transformation of expressions is described in Section 4.3.1. The transformation of statements is described in Section 4.3.2. Function definitions [ES5, Sec. 13] are covered in Section 4.3.2.12. Moreover, hoisting is resolved (see Section 4.3.2.2 and Section 4.3.2.12), identifiers are resolved and renamed to be globally unique, this-bindings are associated with the declaration of the surrounding function and function calls are resolved and inlined (see Section 4.3.2.12).

> **Note**
> Titles of the underlying sections are based on the titles in section 11 and section 12 of the ES5 specification [ES5]. This is done to make it easier to find related sections in the specification.

4.3.1 Expressions

All expressions are transformed so that they are free of side effects. Expressions without side effects are kept as they are (if not stated otherwise). The value of a transformed expression is typically (if not stated otherwise) assigned to a (fresh) variable, whose identifier is then used instead of the original expression. Each description of a transformation states this variable and a sequence (with arbitrary length) of operations expressing the side effects of the expression that have to be added to the CFA. If an expression contains sub-expressions, the operations obtained by the transformation of these sub-expressions are added in the evaluation order of the sub-expressions.

For example, in the expression `a + b` its sub-expression `a` is evaluated first and `b` second. Let

```
c = a + b;
```

be an assignment (expression statement) to a variable called `c` where `a` represents an expression that is transformed to the sequence of operations

```
sideEffectA1;
sideEffectA2;
var resultA = /* ... */
```

with `sideEffectA1` and `sideEffectA2` representing the side-effect operations of `a`, `/* ... */` being an expression introduced by the side-effect transformation, with the result of the transformed expression `a` stored in the (fresh) variable `resultA`, and where `b` represents an expression that is transformed to the sequence of operations

```
sideEffectB1;
var resultB = /* ... */
```

with `sideEffectB1` representing the side-effect operation of `b`, `/* ... */` being an expression introduced by the side-effect transformation, and with the result of the transformed expression `b` stored in the (fresh) variable `resultB`. Then the transformation of `a + b` (which is the right sub-expression of the assignment expression `c = a + b`) results in

```
sideEffectA1;
sideEffectA2;
var resultA = /* ... */
sideEffectB1;
var resultB = /* ... */
var resultExpr = resultA + resultB;
```

where `resultExpr` is the result variable of the expression `a + b`. Consequently, `c = a + b` would be transformed to the sequence of operations

```
sideEffectA1;
sideEffectA2;
var resultA = /* ... */
sideEffectB1;
var resultB = /* ... */
var resultExpr = resultA + resultB;
c = resultExpr;
```

As the auxiliary variable `resultExpr` is referenced only here, it can be eliminated. Thus, the final result of the transformation is the following sequence of operations

```
sideEffectA1;
sideEffectA2;
var resultA = /* ... */
sideEffectB1;
var resultB = /* ... */
c = resultA + resultB;
```

For better readability, we will further omit stating the possible presence of (an arbitrary amount of) side-effect operations preceding to each assignment of a result variable. Instead, we will write in general only, that `a + b` is transformed to

```
var resultA = a;
var resultB = b;
var result = resultA + resultB;
```

where `resultA` is the result variable of transformed expression `a` and `resultB` is the result variable of transformed expression `b`. The statement `var resultA = a` indicates that side-effect operations of the transformation of expression `a` are added at this location, too (that is immediately before the assignment to the result variable `resultA`). Likewise, `var resultB = b;` indicates that side-effect operations of the transformation of expression `b` are added at that location, too (that is immediately before the assignment to the result variable `resultB`). Furthermore, the result of the expression `a + b` will be assigned to a fresh result variable `result` in general.

Note

Sometimes the sequence of operations obtained by transformation may be further simplified. For example, result variables can be eliminated if an expression without side effect is assigned.

The reading of assignable references (identifiers and property accessors) has to be taken into account as a side effect when side effects of sub-expressions are added. For example, in `x + (++x)` the left operand sub-expression `x` is read before the right operand sub-expression `++x` is evaluated. This is important, since `++x` changes the value (increases number value by one) referred by `x`.

Note

Let `x` refer to the value `0`. `x + (++x)` would evaluate the left operand `x` first to `0` resulting in `0 + (++x)`. Then `++x` increases the value of `x` from `0` to `1` and returns the new value `1` as result. So the `+` operator would evaluate `0 + 1` to the result `1`.

The expression `x + (++x)` is not equivalent to

```
var resultIncrement = ++x;
var result = x + resultIncrement;
```

since then `var resultIncrement = ++x;` would change the value referred by `x` before `x` is read in `x + resultIncrement`.

Note

In the example above, let `x` refer to the value `0`. `var resultIncrement = ++x;` evaluates `++x` by increasing the value of `x` from `0` to `1` and returning the new value `1` as result. `var result = x + resultIncrement;` evaluates the left operand `x` as `1` and `resultIncrement` as `1`. This results in evaluation of `1 + 1`, which is `2`. Hence, the result is different to that of `x + (++x)`, which is `1` and not `2`.

Therefore, such a reading of an assignable reference has to be done before the side effects of the following sub-expressions are evaluated. This is achieved by adding an (auxiliary) assignment statement that reads the value of the reference and stores it in a fresh variable. That means, the example `x + (++x)` is transformed to the equivalent statements

```
var oldX = x;
var resultIncrement = ++x;
var result = oldX + resultIncrement;
```

4.3.1.1 Simple Assignment

A simple assignment `lhs = e` regularly adds the side effects of its sub-expressions `lhs` and `e` in evaluation order (`lhs` then `e`). If `lhs` is an identifier, the result is `lhs`. Else, the left hand side `lhs` is a property accessor expression of the form `o.p` or `o[p]`. The result of the transformed (object and property name) sub-expressions is then used in the result property accessor expression instead. For example, `getObject().p = expr` is transformed to

```
var o = getObject();
var e = expr;
o.p = e;
```

and `getObject()[getPropertyName()] = expr` is transformed to

```
var o = getObject();
var p = getPropertyName();
var e = expr;
o[p] = e;
```

where `o`, `p`, and `e` are fresh variables. The result is `o.p` in the first example and `o[p]` in the second example.

Note

`{}.p = expr` is transformed to

```
var o = {};
var e = expr;
o.p = e;
```

Given previous statements

```
var a = { name: 'a' };
var b = { name: 'b' };
var obj = a;
```

the expression `obj.p = (obj = b)` is transformed to

```
var o = obj;
obj = b;
var e = obj;
o.p = e;
```

4.3.1.2 The delete Operator

The delete Operator `delete propertyAccessor` is replaced by a delete operation with side-effect free sub-expression:

```
var pa = propertyAccessor;
delete pa;
```

Due to the assumptions described in Section 4.1, the result value of the expression is always `true` (in strict code).

4.3.1.3 Binary Logical Operators

The expression `left && right` is transformed in the same way as the equivalent if statement

```
var c = left;
if (c) {
    var r = right;
} else {
    var r = c;
}
```

The expression `left || right` is transformed in the same way as the equivalent if statement

```
var c = left;
if (c) {
    var r = c;
} else {
    var r = right;
}
```

In both cases, the result of the expression is stored in the fresh variable `r`. Side effects of the expression `left` are only added once by assigning the evaluated result to a fresh variable `c`.

Note

If `left` has no side effects then the assignment to the variable `c` can be avoided. Just leave the assignment and use `left` instead of `c`.

4.3.1.4 Conditional Operator (? :)

The conditional operator `condition ? thenExpr : elseExpr` is transformed in the same way as the equivalent if statement

```
if (condition) {
    var r = thenExpr;
} else {
    var r = elseExpr;
}
```

in the CFA. Its value is stored in a fresh variable `r`.

4.3.1.5 Prefix Increment Operator

`++x` is transformed in the same way as the equivalent statement `x = (+x) + 1;`. Its value is stored in `x`.

Note

`+x` is used to enforce a conversion of `x` to number.

4.3.1.6 Prefix Decrement Operator

`--x` is transformed in the same way as the equivalent statement `x = x - 1;`. Its value is stored in `x`.

4.3.1.7 Postfix Increment Operator

`x++` is transformed in the same way as the equivalent statements `var r = +x; x = r + 1;`. Its value is stored in a fresh variable `r`.

> **Note**
>
> `+x` is used to enforce a conversion of `x` to number.

4.3.1.8 Postfix Decrement Operator

`x--` is transformed in the same way as the equivalent statements `var r = +x; x = r - 1;`. Its value is stored in a fresh variable `r`.

> **Note**
>
> `+x` is used to enforce a conversion of `x` to number.

4.3.1.9 Compound Assignment

A compound assignment `x @= e` is transformed in the same way as the equivalent statement `x = x @ e;`. `@=` represents one of the assignment operators `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=` or `|=`. `@` represents the respective binary operator of the assignment operator, i.e. `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^` or `|`. For example, `x += 1;` is transformed in the same way as the equivalent statement `x = x + 1;`.

4.3.1.10 Comma Operator

A comma operator `left, right` is transformed in the same way as the equivalent statements `left; var r = right;` where `r` is the fresh result variable of the comma operator expression.

4.3.1.11 Function Expressions

A function expression is transformed by adding a function declaration operation (see Section 4.3.2.12) with a fresh identifier name, which is used as result. For example,

```
func = function () { /* ... */ }
```

is transformed to

```
function f() { /* ... */ }
func = f;
```

where `f` is a fresh identifier name.

4.3.1.12 Function Calls

A CFA is created for each function declaration body. A function CFA has a function entry and exit node. A function call expression results in an edge (with an attached function call operation) from the current node (before the function call) to the function entry node. All operations of the function declaration body statements are appended to the function entry node. A special return variable is introduced (as fresh local variable) for every function declaration. All return statements (see Section 4.3.2.11) lead to the function exit node and assign the return value

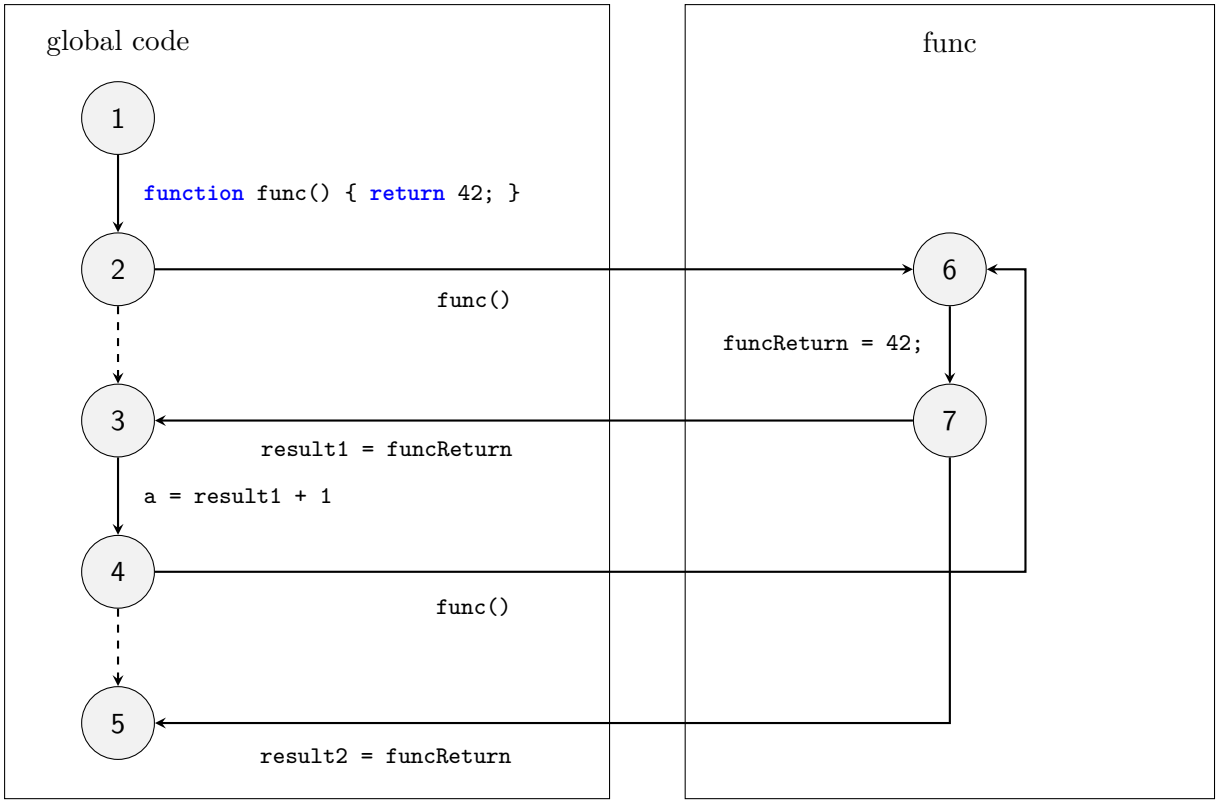


Figure 4.1: CFA of Listing 4.1 that has two function calls to the same function

to this return variable. It is ensured that all paths from the function entry node lead to the function exit node. Therefore, paths without return statement are connected to the function exit node with an edge that assigns `undefined` to the return variable. For every function call edge there is an associated edge (the corresponding function exit edge) that leads from the function exit node to a node representing the location after the function call has been done. To each function exit edge an (assignment) operation is attached that assigns the return variable to the result variable of the function call expression.

If a program contains multiple calls to the same function, of course multiple pairs of function call and exit edges are to be added (one pair for each function call). Figure 4.1 shows the CFA(s) of the program

```
1 function func() { return 42; }
2 a = func() + 1;
3 func();
```

Listing 4.1: Example program with two function calls to the same function

that calls the function `func` twice. The left box contains the CFA (nodes) of the global code (nodes 1 to 5). The right box contains the CFA (nodes) of the function `func` (nodes 6 to 7). The operation of edge 1 → 2 (from node 1 to node 2) declares the function `func`, which is called by the operation `func()` (line 2) of edge 2 → 6. Node 6 is the entry node of function `func`. The edge 6 → 7 shows the operation `funcReturn = 42;` of the transformed `return 42;` statement that assigns the return value to the return variable `funcReturn` of function `func`. Node 7 is the function exit node of function `func`. The edge 7 → 3 (associated with the function call edge 2 → 6) leads back to node 3 after the first function call in the CFA of the global code. Its operation `result1 = funcReturn;` assigns the return value (using the return variable `funcReturn`) of the function `func` to the result variable `result1` of the first function call. This result variable `result1` is then used in the operation `a = result1 + 1` of edge 3 → 4. The function call edge

$4 \rightarrow 6$ represents the second function call (line 3). Its associated function exit edge $7 \rightarrow 5$ leads back to node 5 (after the second function call in the CFA of the global code). Its operation `result2 = funcReturn;` assigns the return value (using the return variable `funcReturn`) of the function `func` to the result variable `result2` of the second function call.

Note

The function exit edge $7 \rightarrow 3$ is associated with the function call edge $2 \rightarrow 6$, whereas the other function exit edge $7 \rightarrow 5$ is associated with the function call edge $4 \rightarrow 6$ of the second function call (line 3). The association is visualized by dashed edges. The dashed edge $2 \rightarrow 3$ indicates that the function call ($2 \rightarrow 6$) finally returns back to node 3. Similarly, the dashed edge $4 \rightarrow 5$ indicates that the function call ($4 \rightarrow 6$) finally returns back to node 5.

A function call expression `expr(e*)` consists of a function (expression) `expr` to be called and an arbitrary amount (list) of expressions `e*` (e.g. `e0, e1, e2`) that are passed as arguments. If `expr` is a function (declaration) identifier, the function is known and the function call edge leads directly to the function entry node of the called function (with `undefined` passed along as `this` argument). Otherwise, the function has to be resolved using the `declaredBy` operator (see Section 4.2.2.1).

We describe the resolution of the function declaration as a special[5] function `callUnknownFunction` that takes `expr` as an extra parameter `functionObject` in addition to the regular parameters. If `expr` is a property accessor expression of the form `o.p` or `o[p]` (where `o` already represents the side-effect free result of the object-expression transformation), `o` is passed as `this` argument. Otherwise, `undefined` is passed as `this` argument. For example, `o.p(1, 'second')` leads to a function call `callUnknownFunction(o.p, 1, 'second')` (with `o` passed as `this` argument).

The function `callUnknownFunction` uses the `declaredBy` operator (see Section 4.2.2.1) to check by which function declaration the passed function object has been declared and delegates the call to the respective function (with the `this` argument of `callUnknownFunction` passed along and `functionObject` as the optional function object argument[6]). For example, if the program contains two function declarations `f` and `g` (that are globally identified), the declaration of `callUnknownFunction` might look similar to

```
function callUnknownFunction(functionObject, p0, p1) {
    if (functionObject declaredBy f) {
        return f(p0, p1);
    } else if (functionObject declaredBy g) {
        return g(p0, p1);
    } else {
        return undefined;
    }
}
```

Listing 4.2: Example of unknown function call resolution function

for a function call `expr(e0, e1)` that is encoded as `callUnknownFunction(expr, e0, e1)`. The last case is reached if an expression is called that is not a function. In this case, an exception would regularly be thrown, but since exceptions are not supported yet (see Section 4.1) `undefined` is returned. Finally, the call `callUnknownFunction(expr, e0, e1)` is inlined by replacing all return statements with assignments to the result variable.

[5] Calls to `callUnknownFunction` are inlined at the end.

[6] The (special) *function object argument* of function call operations is not shown in the code `f(p0, p1)` and `g(p0, p1)` of Listing 4.2. It is required to pass the function instance `functionObject` that provides access to the internal property *[[Scope]]* [ES5, Sec. 15.3.5], which is used to access captured variables (variables in the Lexical Environment [ES5, Sec. 10.2] of the Execution Context [ES5, Sec. 10.3] that has created the function object).

Lets look at a complete example. Listing 4.3 shows a program that contains a function call `o.p(1, 'second')` to an unknown function `o.p`. When it is called, it is not known which function (`f` or `g`) is called. The call is resolved using `callUnknownFunction(o.p, 1, 'second')`, which is inlined as shown in Listing 4.4.

```
function f(p0) { ... }
function g(p0, p1) { ... }
var o = {};
// ...
o.p = f;
// ...
r = o.p(1, 'second'); // callUnknownFunction(o.p, 1, 'second')
```

Listing 4.3: Example of unknown function call

```
function f(p0) { ... }
function g(p0, p1) { ... }
var o = {};
// ...
o.p = f;
// ...
// r = o.p(1, 'second'); // callUnknownFunction(o.p, 1, 'second')
if (functionObject declaredBy f) {
    r = f(1, 'second'); // o passed as this argument and
                        // o.p passed as function object argument
} else if (functionObject declaredBy g) {
    r = g(1, 'second'); // o passed as this argument and
                        // o.p passed as function object argument
} else {
    r = undefined;
}
```

Listing 4.4: Here the unknown function call of Listing 4.3 has been resolved

4.3.1.13 The new Operator

A new operator expression `new expr(e*)` consists of a function (expression) `expr` to be called and an arbitrary amount (list) of expressions `e*` (e.g. `e0, e1, e2`) that are passed as arguments. It is resolved similar to a function call (see Section 4.3.1.12), but using a constructor- instead of a function-call operation. If `expr` is a function (declaration) identifier, the function is known and the constructor call edge leads directly to the function entry node of the called function. Otherwise, the function has to be resolved using the `declaredBy` operator (see Section 4.2.2.1).

We describe the resolution of the function declaration as a special[7] function `callUnknownConstructor` that takes `expr` as an extra parameter `functionObject` in addition to the regular parameters. For example, `new C(1, 'second')` where `C` is a variable identifier leads to a function call `callUnknownConstructor(C, 1, 'second')`. `callUnknownConstructor` uses the `declaredBy` operator to check by which function the passed function object has been declared and delegates the constructor call to the respective function (with `functionObject` as the optional function object argument[8]).

For example, if the program contains two function declarations `f` and `g` (that are globally identified), the declaration of `callUnknownConstructor` might look similar to

[7] Calls to `callUnknownConstructor` are inlined like `callUnknownFunction` at the end.

[8] The (special) *function object argument* of constructor call operations is not shown in the code `new f(p0, p1)` and `new g(p0, p1)` of Listing 4.5. It is required to pass the function instance `functionObject` that provides access to the internal property *[[Scope]]* [ES5, Sec. 15.3.5], which is used to access captured variables (variables in the Lexical Environment [ES5, Sec. 10.2] of the Execution Context [ES5, Sec. 10.3] that has created the function object).

```
function callUnknownConstructor(functionObject, p0, p1) {
    if (functionObject declaredBy f) {
        return new f(p0, p1);
    } else if (functionObject declaredBy g) {
        return new g(p0, p1);
    } else {
        return undefined;
    }
}
```

Listing 4.5: Example of dynamic constructor call resolution function

for a call `new expr(e0, e1)` that is encoded as `callUnknownConstructor(expr, e0, e1)`. The last case is reached if an expression is called that is not a function. In this case, an exception would regularly be thrown, but since exceptions are not supported yet (see Section 4.1) `undefined` is returned. Finally, the call `callUnknownConstructor(expr, e0, e1)` is inlined by replacing all return statements with assignments to the result variable.

4.3.2 Statements

We use the same way of expressing transformations as we used in Section 4.3.1. We will write that a statement

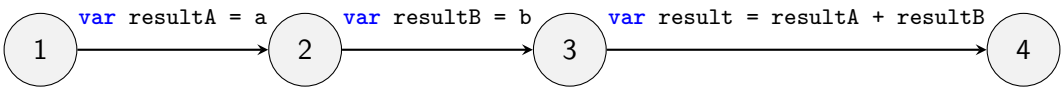
```
var result = a + b
```

is transformed to

```
var resultA = a
var resultB = b
var result = resultA + resultB
```

where `resultA` is the result variable of transformed expression `a` and `resultB` is the result variable of transformed expression `b`. The statement `var resultA = a` indicates that side-effect operations of the transformation of expression `a` are added at this location. Likewise, `var resultB = b` indicates that side-effect operations of the transformation of expression `b` are added at that location.

In case the transformation of a statement can not be expressed using just code, the transformation of the CFA is shown instead. The example above would be displayed as



Here as well, the edge with statement `var resultA = a` indicates that side-effect operations (CFA edges and nodes) of the transformation of expression `a` are added at this location (between node 1 and 2). The same applies to the edge with statement `var resultB = b`.

In very simple cases, we describe the transformation only as text.

4.3.2.1 Block

All statements of the block are added in order to the CFA according to the transformation rule of the respective statement.

4.3.2.2 Variable Statement

The declaration list is split into multiple statements. For example

```
var x = 0, y = 1;
```

is treated like

```
var x = 0;
var y = 1;
```

Declaration and initialization are added to the CFA as separate statements. Hoisting is resolved by moving declarations right behind the entry node of the CFA (before other statements are added). For example

```
var x = 0, y = z;
var z = x;
```

is treated like

```
var x;
var y;
var z;
x = 0;
y = z;
z = x;
```

Each variable is declared only once. For example

```
var x = 0;
var x = 1;
```

is treated like

```
var x;
x = 0;
x = 1;
```

4.3.2.3 Empty Statement

No changes to the CFA are made.

4.3.2.4 Expression Statement

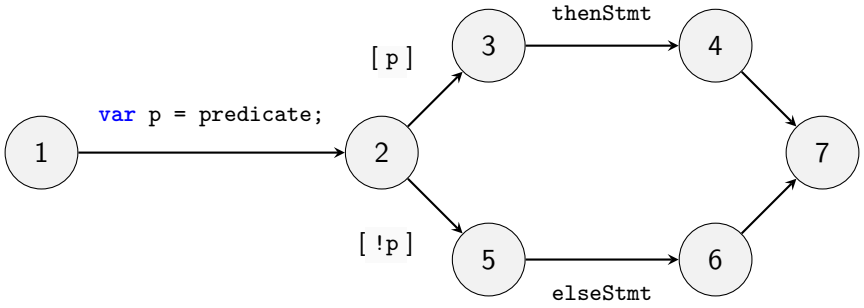
Add all side effects of the expression to the CFA by adding the corresponding operations according to the CFA transformation rules described in Section 4.3.1. If no side effects exist, no changes to the CFA are required.

4.3.2.5 The if Statement

An if statement

```
if (predicate)
    thenStmt;
else
    elseStmt;
```

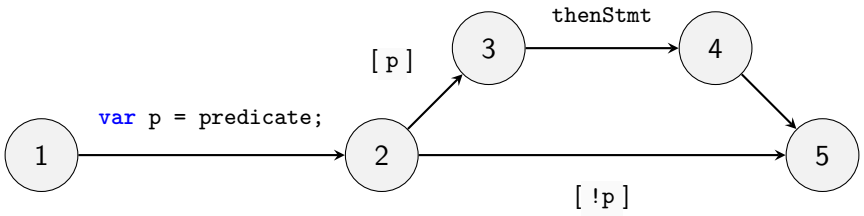
executes the statement `thenStmt` if the specified expression `predicate` is truthy. If the `predicate` is falsy, another statement `elseStmt` is executed. This results in a CFA with two assume edges



where `p` is a fresh variable. If the else case is missing like in

```
if (predicate) thenStmt;
```

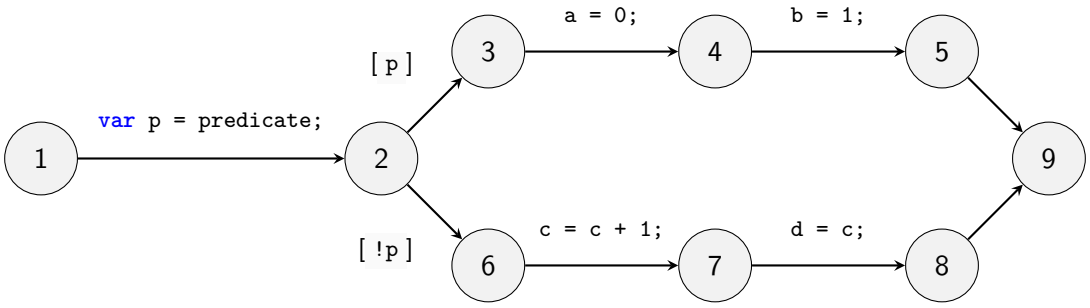
it results in



where `p` is a fresh variable. `thenStmt` and `elseStmt` are shown as simple edges in the CFA, but those statements are added to the CFA branch according to the respective transformation rules of the statement. Thereby, more nodes and edges might be added to the branch. For example

```
if (predicate) {
    a = 0;
    b = 1;
} else
    d = ++c;
```

results in



where `p` is a fresh variable. Side effects of the expression `predicate` are added before the node that is followed by the conditional branching. For example

```
if (x++) y = x;
```

is treated like

```
var tmp = x;
x = x + 1;
if (tmp) y = x;
```

4.3.2.6 The switch Statement

The `switch` and `case` instructions are replaced by semantically equivalent code using if instructions in the CFA. The strict equality operator `===` is used to compare `value` of the `switch(value){...}` instruction with the expressions of the `case` instructions. If the case block contains

- no clauses, then transform the switch statement as the equivalent expression statement `value;`.
- only a clause `default: statement`, then transform the switch statement as the equivalent statements `value; statement;`

- only a clause `case expr: statement`, then transform the switch statement as the equivalent statements

```
if (value === expr) {
    statement;
}
```

- multiple clauses

```
case e0:
    s0;
default:
case e1:
case e2:
    s1;
case e3:
    s2;
```

then transform the switch statement as the equivalent statements

```
v = value
if (v === e0) {
    s0;
    s1;
    s2;
} else if (v === e1 || v === e2) {
    s1;
    s2;
} else if (v === e3) {
    s2;
} else {
    s1;
    s2;
}
```

where `v` is a fresh variable. `v` is used so that `value` is only evaluated once.

Note

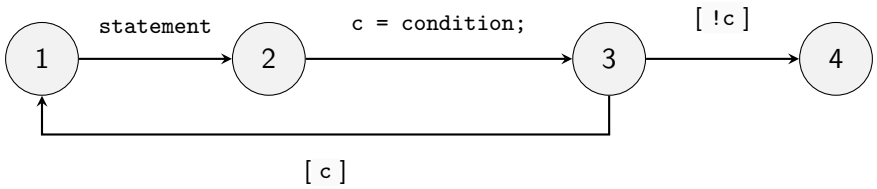
The `||` has to be transformed according to Section 4.3.1.3. The statements `s0`, `s1`, and `s2` might be `break` statements that are transformed as described in Section 4.3.2.10.

Break statements (see Section 4.3.2.10) add edges to specific target nodes. The break target node of a switch statement is the (last) node after all transformed statements (converged).

4.3.2.7 Iteration Statements

The (block) statement to be repeated by an iteration statement may contain continue statements (see Section 4.3.2.9) or break statements (see Section 4.3.2.10) that designate (by their label parameter or by default the closest) one of the surrounding iteration statements (note: nesting may occur). Continue statements and break statements add edges to specific target nodes. Therefore, the description of each iteration statement states also the target node used by continue and by break statements designating it.

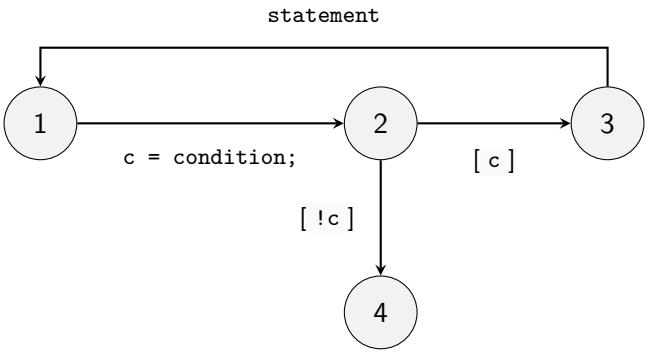
4.3.2.7.1 The do-while Statement A do-while statement `do statement while (condition);` is transformed to the CFA



where `c` is a fresh (declared) variable. Node 2 is the target node of continue statements. Node 4 is the target node of break statements.

4.3.2.7.2 The while Statement

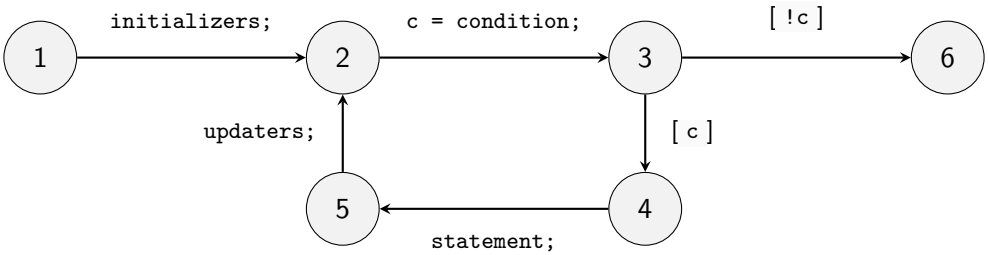
A while statement `while (condition) statement;` is transformed to the CFA



where `c` is a fresh (declared) variable. Node 1 is the target node of continue statements. Node 4 is the target node of break statements.

4.3.2.7.3 The for Statement

A for statement `for (initializers; condition; updaters) statement;` is transformed to the CFA



where `c` is a fresh (declared) variable. Node 5 is the target node of continue statements. Node 6 is the target node of break statements.

4.3.2.8 Labelled Statements

`label: statement;` is encoded in the same way as the equivalent statement `statement;`.

Continue statements (see Section 4.3.2.9) and break statements (see Section 4.3.2.10) add edges to specific target nodes. Continue statements may only refer to labelled iteration statements (see Section 4.3.2.7). If a continue statement refers to `label`, the continue target node of the iteration statement `statement` is used as described in detail in the subsections of Section 4.3.2.7. If a break statement refers to `label`, the break target node of the iteration statement `statement` is used or the node immediately after the operation(s) of `statement` if `statement` is not an iteration statement.

4.3.2.9 The continue Statement

A blank edge (no operation) from the current node (before the continue statement) to the target node of the (labelled) iteration statement (see Section 4.3.2.7) the continue statement refers to is added. Unreachable statements after the continue statement are ignored. Only hoisting of function and variable declarations is resolved (see Section 4.3.2.2 and Section 4.3.2.12).

4.3.2.10 The break Statement

A blank edge (no operation) from the current node (before the break statement) to the target node of the statement the break statement refers to is added. The break statement either refers to an iteration statement (see Section 4.3.2.7), a switch statement (see Section 4.3.2.6), or a labelled statement (see Section 4.3.2.8). Unreachable statements after the break statement are ignored. Only hoisting of function and variable declarations is resolved (see Section 4.3.2.2 and Section 4.3.2.12).

4.3.2.11 The return Statement

A special return variable is introduced (as fresh local variable) for every function declaration as described in Section 4.3.1.12. The return value is assigned to this return variable of the function. If the return statement has no expression to be returned, `undefined` is used instead. The respective assignment operation is attached to an edge that leads from the current node (before the return statement) to the function exit node. Unreachable statements after the return statement are ignored. Only hoisting of function and variable declarations is resolved (see Section 4.3.2.2 and Section 4.3.2.12).

4.3.2.12 Function Declaration

A function declaration is added as operation to the CFA. This operation creates a function object and assigns it to the function identifier of the declaration.

Hoisting is resolved by moving declarations right behind the entry node of the CFA (before other statements are added). However, if the value referred to by the identifier might be changed (for example, a variable declaration with the same identifier name exists or an assignment to the identifier), then the function declaration is renamed (fresh identifier name), a variable declaration with the old name is added (at the declaration section behind the entry node) and the new identifier is assigned to the variable with the old name. For example,

```
function func() { /* ... */ }
func();
func = e;
```

is transformed to

```
function f() { /* ... */ }
var func = f;
func();
func = e;
```

5 SMT Formula Encoding

The formula encoding uses SMT theory of linear integers, arrays [17, 21], floating-point [8, 16, 26], bit-vectors [3, 10], and uninterpreted functions. Most formula encodings are described as a function that is inlined in the overall formula. These functions have names that start with a capital letter to distinguish them from uninterpreted functions and theory operations:

- select(a, i) returns element at index i of array a
- store(a, i, v) stores value v at index i of array a and returns resulting array
- isNaN(n) checks if floating point formula n is NaN
- isZero(n) checks if floating point formula n is $+0$ or -0

Note

We use $=$ in general for equality comparison. Since all comparisons of floating points evaluate to false if either argument is NaN, we write $=_{\text{assign}}$ in case of an assignment of floating points.

5.1 Types

In contrast to statically typed languages, expressions in dynamically typed languages may evaluate to values of different types. Some operations behave differently depending on the type or may implicit convert the value based on its type. For example, the binary addition operator `+` [ES5, Sec. 11.6.1] either performs string concatenation or numeric addition. Therefore, not only the value has to be encoded, but also its type.

5.1.1 Type Tags

Types are encoded similar to the result returned by the `typeof` operator [ES5, Sec. 11.4.3]. That means that a distinction is made between six types:

- The Undefined Type
- The Boolean Type
- The Number Type
- The String Type
- The Object Type
- The Function Type

Note

In contrast to the ES5 specification, the Null Type [ES5, Sec. 8.2] is not seen as a type of its own. Instead the value `null` (see Section 5.2.2) is covered by the object type.

Note

There is no Function Type in the ES5 specification, but we use it for function objects to distinguish them from regular objects as described in Section 5.2.7.

Each type is encoded as a distinct integer that we further refer to as type tag. We will name them for better readability as $\tau_{\text{undefined}}$, τ_{boolean}, τ_{number}, τ_{string}, τ_{object}, and τ_{function}. Thereby, we do not have to remember which integer value is associated with which type.

5.2 Values

There are primitive, object, and function values. This section describes how these values are encoded and what type is associated with them.

5.2.1 Undefined

Since there is only one value of the type Undefined, the formula of the undefined value [ES5, Sec. 4.3.9] does not matter. It is the only value with type tag $\tau_{\text{undefined}}$.

5.2.2 Null

The null value [ES5, Sec. 4.3.11] is encoded as a special object ID (see Section 5.2.6). We use *null* in formulas to refer to this value. The type of the null value is encoded as type tag τ_{object}.

5.2.3 Boolean

There are only two Boolean values, *true* and *false* [ES5, Sec. 4.3.13]. *true* is encoded as boolean formula $\top$. *false* is encoded as boolean formula $\bot$. The type is encoded as type tag τ_{boolean}.

5.2.4 Number

Number values [ES5, Sec. 4.3.19] are encoded as their corresponding floating point formula with an exponent size of 11 and a mantissa size of 52. The type is encoded as type tag τ_{number}.

5.2.5 String

String values [ES5, Sec. 4.3.16] are primitive values in *ECMAScript*. They are immutable finite sequences of characters. Each string value is encoded as a unique (floating point) number that we further refer to as string-ID. We write StringID(`"example"`) in a formula to refer to the string-ID of the string `"example"`. Two string values are encoded as the same string-ID if and only if they are (strict) equal.

Note

Two strings are equal if they are exactly the same sequence of characters (same length and same characters in corresponding positions).

Only string values that appear as string literals or property names in the analyzed program are associated with a (known) string-ID. This also includes values that occur implicitly. For example, each function object has a property with name `prototype`. If a function is declared then the string value `'prototype'` implicitly occurs in the program. The same applies to the `typeof` operator whose return value is either the string value `'undefined'`, `'object'`, `'boolean'`, `'number'`, `'string'`, or `'function'`.

Some operators use (implicit) type conversion. The type conversion of string to number (see Section 5.5.2) and number to string (see Section 5.5.3) requires a mapping of string-IDs that allows these conversions. Therefore, a floating point formula with an exponent size of 12 and a mantissa size of 52 is used for string-IDs. The exponent size 12 is greater than the exponent size 11 of the floating point formula used for numbers (see Section 5.2.4). Thereby, the same (casted) value of a number[1] can be used as string-ID of its string representation [ES5, Sec. 9.8.1]. On the other hand, values outside of the range of number values can be used as string-IDs of non-number strings[2].

String operations are only encoded as uninterpreted function. We do only cover concatenation in form of the binary `+` operator (see Section 5.6.5.1), which is encoded as uninterpreted function $\text{concat}(l, r)$ where l and r are string-IDs. String operations based on String Objects [ES5, Sec. 15.5] are not supported yet since built-in objects are not supported yet as stated in Section 5.7.1. However, they could be encoded in a similar way using respective uninterpreted functions.

5.2.6 Object

An object value is encoded as a unique integer that we further refer to as object-ID. Each object-ID is associated with properties (see Section 5.2.6.1). The null value (see Section 5.2.2) is also represented by a particular object-ID, but is not associated with properties. The type of an object value is τ_{object}.

5.2.6.1 Properties

Properties are managed as an array formula that maps each property name (string-ID formula) to a variable formula (see Section 5.3) that represents the value of the property. If a property is not set on an object, then the property name is mapped to a special variable formula *objectFieldNotSet*, which represents an unset field. *emptyObjectFields* represents the (initial) property mapping, where all property names are mapped to *objectFieldNotSet*.

Properties of an object may change in the course of the program. That means that there is an array formula (property mapping) for each point in time. If a property is changed, a new array formula (updated property mapping) is created based on the old array formula. These array formulas are managed by a statically indexed variable called *objectFields* where the index is used to represent the point in time. Since there may exist multiple objects in a program, the *objectFields* variable itself is an array that maps object-IDs to their current property mapping.

5.2.6.2 Prototype Property

The internal property [[Prototype]] [ES5, Sec. 8.6.2] is encoded as (regular) property with reserved[3] string-ID as name that we refer to by *prototypeField*.

5.2.6.3 Prototype Chain

ECMAScript uses prototype based inheritance for objects. Each object has an internal property [[Prototype]] [ES5, Sec. 8.6.2] which refers to another object called its *prototype*. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, `null` has no prototype, and acts as the final link in this prototype chain.

[1] By adding a zero bit as 12^{th} (HiOrder) bit to the 11bit-exponent of the number's floating point formula.
[2] These are recognizable from the 12^{th} (HiOrder) bit of the exponent of the string-IDs floating point formula being one.
[3] The string-ID is not associated with any string value of the program.

The *prototype chain* comes into play when a property is accessed on an object (see Section 5.6.2.1). If a property is not set on the object, the property is looked up in the prototype of the object. If the property is not set on that prototype, the property is looked up in the prototype of the prototype and so on till a prototype is `null`. As soon as the property is found, it is returned, otherwise `undefined`.

The prototype chain might be arbitrary long but it is always finite. We assume that no prototype chain is longer as a maximum *maxPrototypeChainLength*. Thereby, we can unroll the look-up in the prototype chain (see LookUpOnPrototypeChain in Section 5.6.2.1). The drawback is, that a look-up of a property might return `undefined` if *maxPrototypeChainLength* is too small. This issue is addressed in Section 9.11.

5.2.7 Function

A function object value is encoded like a regular object (see Section 5.2.6), but its type is τ_{function}. Its object-ID is associated with its function declaration (see Section 5.7.3) using an uninterpreted function declarationOf and with its scope (see Section 5.3.2) using an uninterpreted function scopeOf.

5.3 Variables

A variable is encoded as unique integer formula that we further refer to as *variable-ID*. The type of the value stored in a variable is associated with the variable-ID using an uninterpreted function typeof. The value of the variable is also associated using an uninterpreted function. There is one uninterpreted function for each type of value: booleanValue, numberValue, functionValue, objectValue, and stringValue. An assignment `x = 42` is basically encoded as

$$\text{typeof}(x) = \tau_{\text{number}} \wedge \text{numberValue}(x) = 42$$

where x is the variable-ID of `x` and 42 is the floating point formula that represents 42.

5.3.1 Statically Indexed Variables

Similarly to a static single-assignment (SSA) form [11] like it is used by compilers, for each program variable an SSA *index* counter is added that is incremented on every assignment to the variable, and all variable accesses use the variable qualified with the current index value.

For example, the program

```
1 var x = 0;
2 if (predicate) {
3   x = x + 1;
4   x = x * 2;
5 } else {
6   x = x * 3;
7 }
8 x = x - 1;
```

could be interpreted with indexed variables like this

```
1 x0 = 0;
2 if (predicate0) {
3   x1 = x0 + 1;
4   x2 = x1 * 2;
5 } else {
6   x1 = x0 * 3;
7   x2 = x1;
8 }
9 x3 = x2 - 1;
```

whereas the indices of converging branches are merged. After the *then* branch, the index of variable `x` is 2 due to two assignments. The *else* branch only contains one assignment. Thereby, the index of variable `x` would be 1 (different to *then* branch). The indices are merged by adding an assignment x_2 `=` x_1`;` (line 7) at the end of the *else* branch. Thus, the index of variable `x` is 2 in both cases.

5.3.2 Scoped Variables

To resolve an identifier of an *ECMAScript* variable [ES5, Sec. 10.3.1] the *Lexical Environment* [ES5, Sec. 10.2] of the running *Execution Context* [ES5, Sec. 10.3] is needed. If the Lexical Environment has no binding of the identifier, the identifier is looked up in the outer environment reference of the Lexical Environment. This chain of nested environments and the lookup has to be formula encoded. Therefore, each variable is associated with a scope-ID (integer formula) that represents the environment that we call *scope*. This is done using an uninterpreted function $\text{var}(s, x)$ where s is a scope-ID and x is a variable-ID. A unique scope-ID is created for the global code (called *globalScope*) and each function call.

Since environments might be nested, a scope is associated with a *scope stack*, which is an array formula that contains all scopes of the (nested environments) chain. The scope of the environment of the current execution context is encoded as a statically indexed variable that we further refer to as *current scope* or *currentScope* of an execution context. On every function call the current scope has to be updated to encode the change of the running execution context (and of course at the exit of the function call, too). To look up the scope of a declaration, the nesting level of the declaration is used as index on the scope stack of the current scope.

A local variable `x` is encoded as

$$\text{var}(\text{select}(\text{scopeStack}(\mathit{currentScope}), n), x)$$

where n is the nesting level of the declaration of x. A global variable is encoded similar, but since the scope is known to be the global scope, it is simplified to

$$\text{var}(\mathit{globalScope}, x)$$

5.3.2.1 Updating SSA Index Of Scoped Variables

Other scoped variables[4] have to be updated on every assignment to a scoped variable (see Section 5.7.4.1). If a function `f(p)` is called the first time a scope s_0 is created and variables/-parameters are associated with this scope like $\text{var}(s_0, p_0)$. On the second call of `f(p)` another scope s_1 is created and the SSA index of `p` is incremented from 0 to 1. Hence, `p` is associated to s_1 by $\text{var}(s_1, p_1)$. However, if `p` of the first scope is captured in a closure, then it would be addressed by $\text{var}(s_0, p_1)$ instead of $\text{var}(s_0, p_0)$, since the index of variable `p` has changed due to the other call of `f`. To work around, indices of the same variable in other scopes are updated too, when a value is assigned to the variable. Since, `p` is assigned a value on the second call of `f(p)` using $\text{var}(s_1, p_1)$, the index of `p` in s_0 has to be updated by $\text{var}(s_0, p_1) = \text{var}(s_0, p_0)$.

5.4 Type, Value, Kind, and Constraint

The formula encoding of an expression is described by its type (tag), value, and kind. The value is either a (simple) value as described in Section 5.2 or a variable-ID as described in Section 5.3. Variables are represented by a variable-ID that may be associated with any value described in Section 5.2. Variables must be treated differently in expressions than simple values, since a

[4] The same variable declaration, but a different scope-ID.

different uninterpreted function must be used for each type to get the simple value associated with the variable. Therefore, the formula encoding of an expressions is not only described by its type and value, but also by its kind, which indicates if the (encoded) value is a simple value or a variable-ID.

Additionally, the formula encoding of some expressions requires further conditions (boolean formulas) that have to apply in the formula of the operation the expression appears in[5]. We call these conditions *constraints*. For example, an Array Initialiser (see Section 5.6.1.4) has an object-ID o as value that needs to be marked as Array using a constraint isArray(o).

5.5 Type Conversion

Operators of expressions (see Section 5.7) perform automatic type conversion as needed. This section describes the conversion functions that are similar to the abstract operations of the ES5 specification. [ES5, Sec. 9] Each type conversion is defined as a function

$$\text{ToType}(e) := \text{ToType}_{\text{Kind}(e)}(\text{Type}(e), \text{Value}(e))$$

where

- e is an expression,
- Type(e) represents the type of e,
- Value(e) represents the value of e,
- Kind(e) represents the kind of e (i.e. value or variable),

Hence, there are two definitions $\text{ToType}_{\text{variable}}(t, v)$ and $\text{ToType}_{\text{value}}(t, v)$. $\text{ToType}_{\text{variable}}(t, v)$ is used if v is a variable-ID and $\text{ToType}_{\text{value}}(t, v)$ is used if v is a value.

Note

Some functions are described using (sub-) function definitions that are named *FromType*To*TargetType*(x). *FromType* is the name of the type that should be converted to the type named *TargetType*. x is a value of type *FromType*. For example, NumberToString(n) describes the type conversion of a number value n to a string value.

5.5.1 ToBoolean

This function is similar to the abstract operation *ToBoolean* of the ES5 specification [ES5, Sec. 9.2].

$$\text{ToBoolean}_{\text{value}}(t, v) := \begin{cases} v & t = \tau_{\text{boolean}} \\ \top & t = \tau_{\text{function}} \\ \text{NumberToBoolean}(v) & t = \tau_{\text{number}} \\ v \neq \mathit{null} & t = \tau_{\text{object}} \\ \text{StringToBoolean}(v) & t = \tau_{\text{string}} \\ \bot & t = \tau_{\text{undefined}} \end{cases}$$

[5] Therefore, these conditions are added to the constraints of the operation (see Section 5.7).

$$\text{ToBoolean}_{\text{variable}}(t, v) := \begin{cases} \text{booleanValue}(v) & t = \tau_{\text{boolean}} \\ \top & t = \tau_{\text{function}} \\ \text{NumberToBoolean}(\text{numberValue}(v)) & t = \tau_{\text{number}} \\ \text{objectValue}(v) \neq \mathit{null} & t = \tau_{\text{object}} \\ \text{StringToBoolean}(\text{stringValue}(v)) & t = \tau_{\text{string}} \\ \bot & t = \tau_{\text{undefined}} \end{cases}$$

$$\text{NumberToBoolean}(n) := \neg\,\text{isZero}(n) \wedge \neg\,\text{isNaN}(n)$$

$$\text{StringToBoolean}(s) := s \neq \text{StringID}(\texttt{""})$$

5.5.2 ToNumber

This function is similar to the abstract operation *ToNumber* of the ES5 specification [ES5, Sec. 9.3].

$$\text{ToNumber}_{\text{value}}(t, v) := \begin{cases} \text{BooleanToNumber}(v) & t = \tau_{\text{boolean}} \\ v & t = \tau_{\text{number}} \\ \text{StringToNumber}(v) & t = \tau_{\text{string}} \\ \text{ObjectToNumber}(v) & t = \tau_{\text{object}} \\ \mathit{NaN} & \text{else} \end{cases}$$

$$\text{ToNumber}_{\text{variable}}(t, v) := \begin{cases} \text{BooleanToNumber}(\text{booleanValue}(v)) & t = \tau_{\text{boolean}} \\ \text{numberValue}(v) & t = \tau_{\text{number}} \\ \text{StringToNumber}(\text{stringValue}(v)) & t = \tau_{\text{string}} \\ \text{ObjectToNumber}(\text{objectValue}(v)) & t = \tau_{\text{object}} \\ \mathit{NaN} & \text{else} \end{cases}$$

$$\text{BooleanToNumber}(b) := \begin{cases} 1 & b \\ 0 & \neg b \end{cases}$$

$$\text{ObjectToNumber}(o) := \begin{cases} 0 & o = \mathit{null} \\ \mathit{NaN} & \text{else} \end{cases}$$

$$\text{StringToNumber}(s) := \begin{cases} 0 & s = \text{StringID}(\texttt{""}) \\ \text{CastStringToNumber}(s) & \text{IsNumberString}(s) \\ \mathit{NaN} & \text{else} \end{cases}$$

Note

StringToNumber covers not all cases described in the ES5 specification [ES5, Sec. 9.3.1], but at least all string representations of numbers.

IsNumberString(s) is $\top$ if string-ID (floating point) formula s is in range of floating point type of number formula, otherwise $\bot$. CastStringToNumber(s) casts string-ID (floating point) formula s to floating point type of number formula.

5.5.3 ToString

This function is similar to the abstract operation *ToString* of the ES5 specification [ES5, Sec. 9.8].

$$\text{ToString}_{\text{value}}(t, v) := \begin{cases} \text{BooleanToString}(v) & t = \tau_{\text{boolean}} \\ \text{FunctionToString}(v) & t = \tau_{\text{function}} \\ \text{NumberToString}(v) & t = \tau_{\text{number}} \\ \text{ObjectToString}(v) & t = \tau_{\text{object}} \\ v & t = \tau_{\text{string}} \\ \text{StringID}(\texttt{"undefined"}) & t = \tau_{\text{undefined}} \end{cases}$$

$$\text{ToString}_{\text{variable}}(t, v) := \begin{cases} \text{BooleanToString}(\text{booleanValue}(v)) & t = \tau_{\text{boolean}} \\ \text{FunctionToString}(\text{functionValue}(v)) & t = \tau_{\text{function}} \\ \text{NumberToString}(\text{numberValue}(v)) & t = \tau_{\text{number}} \\ \text{ObjectToString}(\text{objectValue}(v)) & t = \tau_{\text{object}} \\ \text{stringValue}(v) & t = \tau_{\text{string}} \\ \text{StringID}(\texttt{"undefined"}) & t = \tau_{\text{undefined}} \end{cases}$$

$$\text{BooleanToString}(b) := \begin{cases} \text{StringID}(\texttt{"true"}) & b \\ \text{StringID}(\texttt{"false"}) & \neg b \end{cases}$$

$$\text{FunctionToString}(f) := \text{unknownStringID}(f)$$

$$\text{NumberToString}(n) := \begin{cases} \text{StringID}(\texttt{"NaN"}) & \text{isNaN}(n) \\ \text{StringID}(\texttt{"-Infinity"}) & n = -\infty \\ \text{StringID}(\texttt{"Infinity"}) & n = \infty \\ \text{cast } n \text{ to floating point type of string formula} & \text{else} \end{cases}$$

$$\text{ObjectToString}(o) := \text{unknownStringID}(o)$$

unknownStringID is an uninterpreted function that maps an object-ID to the string-ID of the (not statically known) string value of the object.

Note

A function-ID is also an object-ID.

5.5.4 ToObject

This function is similar to the abstract operation *ToObject* of the ES5 specification [ES5, Sec. 9.9], whereas Undefined and Null can not be handled regularly, since errors are not supported yet. Instead the value is represented by a variable *unknownObjectId*. The same applies to Boolean, Number, and String objects, since these built-in objects are not supported yet.

$$\text{ToObject}_{\text{value}}(t, v) := \begin{cases} v & t = \tau_{\text{function}} \vee t = \tau_{\text{object}} \\ \mathit{unknownObjectId} & \text{else} \end{cases}$$

$$\text{ToObject}_{\text{variable}}(t, v) := \begin{cases} \text{objectValue}(v) & t = \tau_{\text{function}} \vee t = \tau_{\text{object}} \\ \mathit{unknownObjectId} & \text{else} \end{cases}$$

5.5.5 ToFunction

There is no similar abstract operation described in the ES5 specification. Everything except functions is converted to an unknown function ID represented by a variable *notAFunctionId*.

$$\text{ToFunction}_{\text{value}}(t, v) := \begin{cases} v & t = \tau_{\text{function}} \\ \mathit{notAFunctionId} & \text{else} \end{cases}$$

$$\text{ToFunction}_{\text{variable}}(t, v) := \begin{cases} \text{functionValue}(v) & t = \tau_{\text{function}} \\ \mathit{notAFunctionId} & \text{else} \end{cases}$$

5.5.6 ToInt32

ToInt32 converts its argument to one of 2^{32} integer values (encoded as (signed) bit-vector with fixed length of 32) in the range -2^{31} through 2^{31-1} (inclusive) as desribed in the ES5 specifictation [ES5, Sec. 9.5]. It only accepts a number formula (see Section 5.2.4). Hence, ToNumber is used to convert the argument to a number formula. *NaN* and $\pm\infty$ result in the bit-vector (with fixed-length of 32) formula of zero. Otherwise, the number formula is rounded toward zero (step 3 in the specification) to a bit-vector with fixed-length 1026 that is used in the operations described in step 4 and 5 in the specification. The result is then converted to a bit-vector formula with fixed-length of 32 by extracting the respective bits.

5.5.7 ToUint32

ToUint32 converts its argument to one of 2^{32} integer values (encoded as (unsigned) bit-vector with fixed length of 32) in the range 0 through 2^{32-1} (inclusive) as desribed in the ES5 specifictation [ES5, Sec. 9.6]. It only accepts a number formula (see Section 5.2.4). Hence, ToNumber is used to convert the argument to a number formula. *NaN*, ± 0, and $\pm\infty$ result in the bit-vector (with fixed-length of 32) formula of zero. Otherwise, the number formula is rounded toward zero (step 3 in the specification) to a bit-vector with fixed-length 1026 that is used in the operations described in step 4 and 5 in the specification. The result is then converted to a bit-vector formula with fixed-length of 32 by extracting the respective bits.

5.6 Expressions

This section describes the formula encoding of Expressions [ES5, Sec. 11] that are part of the CFA (see Section 4.3.1) as well as the `declaredBy` expression (see Section 4.2.2.1). Each expression encoding is described by a value, a type, and a kind (variable or value).

We don't mention type or/and kind if it's evident from the value. If the value of an expression is described as a specific simple value, then its type is the respective type of that simple value (see Section 5.2) and its kind is *value*. On the other hand, if the value of an expression is described as a variable-ID x, then its type is typeof(x) and its kind is *variable*. If it is described as a statically indexed variable x, we (usually) only mention the variable (name). In this case, the value is the variable-ID x_i where i is the current index counter of x.

Note

We use Type(e) in formulas (that describe values) to refer to the type of the expression e.

Note

Titles of the underlying sections are based on the titles in section 11 of the ES5 specification [ES5]. This is done to make it easier to find related sections in the specification.

5.6.1 Primary Expressions

This section describes the formula encoding of Primary Expressions [ES5, Sec. 11.1] that are part of the CFA.

5.6.1.1 The this Keyword

`this` is encoded as the this-variable of the current function (declaration). A (statically indexed) this-variable exists for each function declaration. They are indexed independently of each other and are initialized by a function call (see Section 5.7.6) or constructor call (see Section 5.7.7).

5.6.1.2 Identifier Reference

Let x_i be the statically indexed identifier (see Section 5.3.1) where `x` is the identifier name and i is the current index counter of `x`. If `x` is a (global) identifier (see Section 4.2.3), it is either a (predefined) property of the global object [ES5, Sec. 15.1] or an undeclared global variable (introduced by the program). Otherwise, it is a declared (global or local) variable.

5.6.1.2.1 Declared Variable A local variable is encoded as a scoped variable (see Section 5.3.2)

$$\text{var}(\text{select}(\text{scopeStack}(\mathit{currentScope}), n), x_i)$$

where n is the nesting level of the declaration of x_i. A global variable is encoded similar, but since the scope is known to be the global scope, it is simplified to

$$\text{var}(\mathit{globalScope}, x_i)$$

5.6.1.2.2 Undeclared Global Variable If i of an undeclared global variable is the initial static index (see Section 5.3.1), then no value has been assigned and the result is encoded as the undefined value (see Section 5.2.1). Otherwise, it is treated like a declared global variable. That means $\text{var}(\mathit{globalScope}, x_i)$.

5.6.1.2.3 Properties Of The Global Object The value properties [ES5, Sec. 15.1.1] `NaN`, `Infinity` and `undefined` are encoded as their respective values (see Section 5.2). We do not cover other predefined identifiers (properties), but they can be encoded in a similar way by their value.

Note

Function values (for example `isNaN` [ES5, Sec. 15.1.2.4] or *Constructor Properties of the Global Object* [ES5, Sec. 15.1.4] like `Boolean`) require a special treatment in function-/constructor-call operations (see Section 5.7).

All uncovered (predefined) identifiers are treated like undeclared global variables that usually (if no value has been assigned) result in the undefined value (see Section 5.2.1).

5.6.1.3 Literal Reference

Null-, Boolean-, Numeric- and String-Literals are encoded as their respective values as described in Section 5.2.

5.6.1.4 Array Initialiser

An Array Initialiser `[e0, e1, e2]` is encoded equivalent to an Object Initialiser (see Section 5.6.1.5) `{'0': e0, '1': e1, '2': e2, length: 3 }`. However, the object-ID o is marked as an array using a constraint $\text{isArray}(o)$.

5.6.1.5 Object Initialiser

Type is τ_{object}. Value is a fresh object-ID o. A constraint $\neg\,\text{isArray}(o)$ marks the object to be not an array (in contrast to an Array Initialiser (see Section 5.6.1.4)). The index counter of the object fields variable *objectFields* is increased from its *old* index to its *new* index

$$objectFields_{new} = \text{store}(objectFields_{old}, o, fields)$$

where *fields* is object-fields array of o. *fields* is based on *emptyObjectFields* to which all properties of the object initialiser have been assigned to as described in Section 5.7.4.2.1, whereas it is only required to update (the index of) *objectFields* once. For example,

- *fields* of `{}` (object initialiser without any properties) is *emptyObjectFields*
- *fields* of `{ p0: e0, "p1": e1 }` is

  $$\text{store}(\text{store}(emptyObjectFields, p_0, e_0), p_1, e_1)$$

  where p_0 is the property name (string-ID) of `p0`, e_0 is the (set) property variable (of p_0) with `e0` assigned to it, p_1 is the property name (string-ID) of `"p1"` (that means StringID(`"p1"`)) and e_1 is the (set) property variable (of p_1) with `e1` assigned to it.

Note

emptyObjectFields is described in Section 5.2.6.1.

5.6.1.6 The Grouping Operator

The Grouping Operator `(e)` is encoded like `e`.

5.6.2 Left-Hand-Side Expressions

This section describes the formula encoding of Left-Hand-Side Expressions [ES5, Sec. 11.2] that are part of the CFA, whereas only Property Accessors [ES5, Sec. 11.2.1] remain as expressions in the CFA.

Note

The new Operator [ES5, Sec. 11.2.2] appears as constructor call operation (in the CFA), whose formula encoding is described in Section 5.7.7. Function Calls [ES5, Sec. 11.2.3] appear as function call operations (in the CFA), whose formula encoding is described in Section 5.7.6. Argument Lists [ES5, Sec. 11.2.4] are part of function/constructor call operations in the CFA. Function Expressions [ES5, Sec. 11.2.5] have been transformed to function declaration operations (in the CFA), whose formula encoding is described in Section 5.7.3.

Note

The title of this section is adopted from the ES5 specification. It may give the impression that only Property Accessors appears as left hand sides in assignment operations, but identifier references (see Section 5.6.1.2) are also allowed.

5.6.2.1 Property Accessors

Property Accessors [ES5, Sec. 11.2.1] exist in two different notations that have to be handled slightly different, since bracket notation has to consider array indices.

Note

Dot notation has an identifier as property name, which can not be an array index (string), since identifiers must not begin with a digit [ES5, Sec. 7.6].

Note

This section only describes the read access. The write access is covered in Section 5.7.4.2.

5.6.2.1.1 Dot Notation A Property Accessor in dot notation `obj.propName` consists of an expression `obj` and an identifier name `propName`. The identifier name is used as string value (here StringID(`"propName"`)). The Property Accessor is encoded as the variable returned by AccessField(ToObject(obj), StringID(`"propName"`)).

$$\text{AccessField}(o, p) := \begin{cases} \mathit{fieldOnPrototype} & \mathit{field} = \mathit{objectFieldNotSet} \\ \mathit{field} & \text{else} \end{cases}$$

where

$$\begin{aligned} \mathit{fields} &:= \text{GetObjectFields}(o) \\ \mathit{field} &:= \text{select}(\mathit{fields}, p) \\ \mathit{fieldOnPrototype} &:= \text{LookUpOnPrototypeChain}(1, \text{select}(\mathit{fields}, \mathit{prototypeField}), p) \end{aligned}$$

Note

prototypeField is defined in Section 5.2.6.2.

$$\text{GetObjectFields}(o) := \text{select}(\mathit{objectFields}_i, o)$$

where o is an object-ID and i is the current index of the statically indexed variable *objectFields*.

LookUpOnPrototypeChain(d, $\mathit{protoVar}$, p) recursively looks up the property on the prototype chain of an object. d is the depth (integer) of the current prototype in the prototype chain. The current prototype is passed as variable $\mathit{protoVar}$. p is the string-ID of the property to look up.

$$\text{LookUpOnPrototypeChain}(d, \mathit{protoVar}, p) := \begin{cases} \mathit{undefined} & d > \mathit{maxPrototypeChainLength} \\ & \quad \vee\ \mathit{protoVar} = \mathit{objectFieldNotSet} \\ \mathit{parentPrototype} & \mathit{isFieldOnPrototypeNotSet} \\ \mathit{fieldOnPrototype} & \text{else} \end{cases}$$

where *undefined* is a variable with $\text{typeof}(\mathit{undefined}) = \tau_{\text{undefined}}$ and where

$$
\begin{aligned}
\mathit{prototypeFields} &:= \text{GetObjectFields}(\text{objectValue}(\mathit{protoVar})) \\
\mathit{fieldOnPrototype} &:= \text{select}(\mathit{prototypeFields}, p) \\
\mathit{isFieldOnPrototypeNotSet} &:= (\mathit{fieldOnPrototype} = \mathit{objectFieldNotSet}) \\
\mathit{parentPrototype} &:= \text{LookUpOnPrototypeChain}(d+1, \text{select}(\mathit{prototypeFields}, \mathit{prototypeField}), p)
\end{aligned}
$$

Note

maxPrototypeChainLength is described in Section 5.2.6.3 and resolves the recursion. LookUpOnPrototypeChain could not be inlined without it.

5.6.2.1.2 Bracket Notation A Property Accessor in bracket notation `obj[propExpr]` consists of expressions `obj` and `propExpr`. The Property Accessor is encoded as the variable returned by BracketPropertyAccess(ToObject(obj), ToString($propExpr$)).

$$
\text{BracketPropertyAccess}(o, p) := \begin{cases} \mathit{undefined} & \mathit{isUndefinedArrayElementIndex} \\ \text{select}(o, p) & \text{else} \end{cases}
$$

where

$$
\begin{aligned}
\mathit{isUndefinedArrayElementIndex} &:= \text{isArray}(o) \wedge \text{IsArrayIndexString}(p) \\
&\quad \wedge \text{ToNumber}_{\text{value}}(p) \geq \text{ToNumber}(\mathit{lengthProperty}) \\
\mathit{lengthProperty} &:= \text{AccessField}(o, \text{StringID}(\texttt{"length"}))
\end{aligned}
$$

IsArrayIndexString(s) checks if s is an array index [ES5, Sec. 15.4]. That means a nonnegative integer less than 2^{32}.

5.6.3 Unary Operators

All unary operators have an expression as operand. The result is always a value and not a variable.

5.6.3.1 The void Operator

`void op` is encoded as the undefined value (see Section 5.2.1).

5.6.3.2 The typeof Operator

Type is τ_{string}. Value of `typeof op` is TypeOf(Type(op)).

$$
\text{TypeOf}(t) := \begin{cases}
\text{StringID}(\texttt{"boolean"}) & t = \tau_{\text{boolean}} \\
\text{StringID}(\texttt{"function"}) & t = \tau_{\text{function}} \\
\text{StringID}(\texttt{"number"}) & t = \tau_{\text{number}} \\
\text{StringID}(\texttt{"string"}) & t = \tau_{\text{string}} \\
\text{StringID}(\texttt{"undefined"}) & t = \tau_{\text{undefined}} \\
\text{StringID}(\texttt{"object"}) & \text{else}
\end{cases}
$$

5.6.3.3 Unary + Operator

Type is τ_{number}. Value of `+op` is ToNumber(op).

5.6.3.4 Unary - Operator

Type is τ_{number}. Value of `-op` is $-$ ToNumber(op).

5.6.3.5 Bitwise NOT Operator ($\sim$)

Type is τ_{number}. The operand is casted to bit-vector formula using ToInt32 to be able to use the complement operator of bit-vector theory. The result is casted back to a floating-point number (see Section 5.2.4) and represents the value.

5.6.3.6 Logical NOT Operator (!)

Type is τ_{boolean}. Value of `!op` is $\neg$ ToBoolean(op).

5.6.4 Multiplicative Operators

Type is τ_{number}. Both operands are converted using ToNumber. The respective floating point theory operators are used for multiplication, division, and remainder.

5.6.5 Additive Operators

This section describes the formula encoding of Additive Operators [ES5, Sec. 11.6].

5.6.5.1 The Addition operator (+)

The Addition operator `l + r` either performs string concatenation or numeric addition. The result depends on the types of the operands:

$$\mathit{resultIsString} := \text{Type}(l) = \tau_{\text{string}} \vee \text{Type}(r) = \tau_{\text{string}}$$

The result of the string concatenation is encoded using an uninterpreted function concat

$$\mathit{concatResult} := \text{concat}(\text{ToString}(l), \text{ToString}(r))$$

The result of the numeric addition is encoded as

$$\mathit{numericAdditionResult} := \text{ToNumber}(l) + \text{ToNumber}(r)$$

Since the value might either be a string or number, the result is encoded as a fresh variable *result*. The variable is associated with its type using a constraint

$$\text{typeof}(\mathit{result}) = \begin{cases} \tau_{\text{string}} & \mathit{resultIsString} \\ \tau_{\text{number}} & \text{else} \end{cases}$$

and its value using a constraint *valueConstraint* defined as

$$\mathit{valueConstraint} := \begin{cases} \text{stringValue}(\mathit{result}) = \mathit{concatResult} & \mathit{resultIsString} \\ \text{numberValue}(\mathit{result}) =_{\text{assign}} \mathit{numericAdditionResult} & \text{else} \end{cases}$$

5.6.5.2 The Subtraction operator (-)

Type is τ_{number}. Value of `l - r` is ToNumber(l) $-$ ToNumber(r).

5.6.6 Bitwise Shift Operators

This section describes the formula encoding of Bitwise Shift Operators [ES5, Sec. 11.7].

5.6.6.1 The Left Shift Operator (`<<`)

Type is τ_{number}. The formula encoding is similar to the description of The Left Shift Operator [ES5, Sec. 11.7.1]. Operands are casted to bit-vector formula using ToInt32 (see Section 5.5.6) or ToUint32 (see Section 5.5.7) to be able to use the left-shift operator of bit-vector theory. The result is casted back to a floating-point number (see Section 5.2.4) and represents the value.

5.6.6.2 The Signed Right Shift Operator (`>>`)

Type is τ_{number}. The formula encoding is similar to the description of The Signed Right Shift Operator [ES5, Sec. 11.7.2]. Operands are casted to bit-vector formula using ToInt32 (see Section 5.5.6) or ToUint32 (see Section 5.5.7) to be able to use the signed-right-shift operator of bit-vector theory. The result is casted back to a floating-point number (see Section 5.2.4) and represents the value.

5.6.6.3 The Unsigned Right Shift Operator (`>>>`)

Type is τ_{number}. The formula encoding is similar to the description of The Unsigned Right Shift Operator [ES5, Sec. 11.7.3]. Operands are casted to bit-vector formula using ToInt32 (see Section 5.5.6) or ToUint32 (see Section 5.5.7) to be able to use the unsigned-right-shift operator of bit-vector theory. The result is casted back to a floating-point number (see Section 5.2.4) and represents the value.

5.6.7 Relational Operators

This section describes the formula encoding of Relational Operators [ES5, Sec. 11.8]. As mentioned in Section 4.1, we assume that relational operators `<`, `>`, `<=`, and `>=` do not compare string values. Hence, we always[6] convert operands to numbers according to step 3 of the Abstract Relational Comparison Algorithm [ES5, Sec. 11.8.5].

5.6.7.1 The Less-than Operator (`<`)

Type is τ_{boolean}. Value of `l < r` is $\text{ToNumber}(l) < \text{ToNumber}(r)$.

5.6.7.2 The Greater-than Operator (`>`)

Type is τ_{boolean}. Value of `l > r` is $\text{ToNumber}(l) > \text{ToNumber}(r)$.

5.6.7.3 The Less-than-or-equal Operator (`<=`)

Type is τ_{boolean}. Value of `l <= r` is $\text{ToNumber}(l) \leq \text{ToNumber}(r)$.

5.6.7.4 The Greater-than-or-equal Operator (`>=`)

Type is τ_{boolean}. Value of `l >= r` is $\text{ToNumber}(l) \geq \text{ToNumber}(r)$.

[6] Normally, step 4 of the Abstract Relational Comparison Algorithm [ES5, Sec. 11.8.5] would apply if both operands are strings. However, our formula encoding of strings is not precise enough to handle this case adequately.

5.6.7.5 The instanceof operator

Type is τ_{boolean}. Value of `l instanceof r` is HasInstance(ToObject(l), ToObject(r)). HasInstance checks if `r.prototype` exists as *prototypeField* on the prototype chain of `l`.

$$\text{HasInstance}(i, o) := \text{FindPrototype}(1, proto, instanceProto)$$

where instance i and object o are object-IDs and

$$proto := \text{AccessField}(o, \text{StringID}(\texttt{"prototype"}))$$
$$instanceProto := \text{select}(\text{GetObjectFields}(i), prototypeField)$$

Note

prototypeField is defined in Section 5.2.6.2. AccessField and GetObjectFields are defined in Section 5.6.2.1.1.

FindPrototype(d, p, i) recursively looks up if the object p (property variable-ID) exists as *prototypeField* on the prototype chain of the instance. d is the depth (integer) of the current prototype i in the prototype chain. The current prototype property of the instance is passed as (property) variable-ID i.

$$\text{FindPrototype}(d, p, i) := \begin{cases} \bot & d > maxPrototypeChainLength \\ & \quad \vee\ i = objectFieldNotSet \\ \top & \text{objectValue}(p) = \text{objectValue}(i) \\ foundOnParent & \text{else} \end{cases}$$

where

$$instanceProto := \text{select}(\text{GetObjectFields}(\text{objectValue}(i)), prototypeField)$$
$$foundOnParent := \text{FindPrototype}(d + 1, p, instanceProto)$$

Note

maxPrototypeChainLength is described in Section 5.2.6.3 and resolves the recursion. FindPrototype could not be inlined without it.

5.6.7.6 The in operator

Type is τ_{boolean}. Value of `l in r` is HasProperty(ToString(l), ToObject(r)).

$$\text{HasProperty}(p, o) := \begin{cases} isSetOnPrototype & field = objectFieldNotSet \\ \top & \text{else} \end{cases}$$

where

$$
\begin{aligned}
\mathit{fields} &:= \text{GetObjectFields}(o) \\
\mathit{field} &:= \text{select}(\mathit{fields}, p) \\
\mathit{isSetOnPrototype} &:= \text{PrototypeHasProperty}(1, \text{select}(\mathit{fields}, \mathit{prototypeField}), p)
\end{aligned}
$$

Note

prototypeField is defined in Section 5.2.6.2. GetObjectFields is defined in Section 5.6.2.1.1.

PrototypeHasProperty($d, \mathit{protoVar}, p$) recursively looks up if the property exists on the prototype chain of an object. d is the depth (integer) of the current prototype in the prototype chain. The current prototype is passed as variable $\mathit{protoVar}$. p is the string-ID of the property to be looked up.

$$
\text{PrototypeHasProperty}(d, \mathit{protoVar}, p) := \begin{cases} \bot & d > \mathit{maxPrototypeChainLength} \\ & \quad \vee \mathit{protoVar} = \mathit{objectFieldNotSet} \\ \mathit{isSetOnParentPrototype} & \mathit{isNotSetOnPrototype} \\ \top & \text{else} \end{cases}
$$

where

$$
\begin{aligned}
\mathit{prototypeFields} &:= \text{GetObjectFields}(\text{objectValue}(\mathit{protoVar})) \\
\mathit{fieldOnPrototype} &:= \text{select}(\mathit{prototypeFields}, p) \\
\mathit{isNotSetOnPrototype} &:= (\mathit{fieldOnPrototype} = \mathit{objectFieldNotSet}) \\
\mathit{isSetOnParentPrototype} &:= \text{PrototypeHasProperty}(d+1, \text{select}(\mathit{prototypeFields}, \mathit{prototypeField}), p)
\end{aligned}
$$

Note

maxPrototypeChainLength is described in Section 5.2.6.3 and resolves the recursion. PrototypeHasProperty could not be inlined without it.

5.6.8 Equality Operators

This section describes the formula encoding of Equality Operators [ES5, Sec. 11.9].

5.6.8.1 The Equals Operator (`==`)

Type is τ_{boolean}. Value of `l == r` is Equals(l, r). The encoding is similar to *The Abstract Equality Comparison Algorithm* [ES5, Sec. 11.9.3] without the cases (8 and 9) that use *ToPrimitive*, which is not supported yet due to an implicit function call of an internal method (see Section 4.1). In these cases, the else case (10) applies.

$$\text{Equals}(l,r) := \begin{cases} \text{SameTypeEquals}(l,r) & \text{Type}(l) = \text{Type}(r) \\ \top & \text{IsNull}(l) \wedge \text{Type}(r) = \tau_{\text{undefined}} \\ \top & \text{Type}(l) = \tau_{\text{undefined}} \wedge \text{IsNull}(r) \\ \text{Equals}(l, \text{ToNumber}(r)) & \text{Type}(l) = \tau_{\text{number}} \wedge \text{Type}(r) = \tau_{\text{string}} \\ \text{Equals}(\text{ToNumber}(l), r) & \text{Type}(l) = \tau_{\text{string}} \wedge \text{Type}(r) = \tau_{\text{number}} \\ \text{Equals}(\text{ToNumber}(l), r) & \text{Type}(l) = \tau_{\text{boolean}} \wedge \text{Type}(r) = \tau_{\text{number}} \\ \text{Equals}(l, \text{ToNumber}(r)) & \text{Type}(l) = \tau_{\text{number}} \wedge \text{Type}(r) = \tau_{\text{boolean}} \\ \bot & \text{else} \end{cases}$$

$$\text{IsNull}(e) := \text{Type}(e) = \tau_{\text{object}} \wedge \text{ToObject}(e) = \mathit{null}$$

SameTypeEquals handles case 1 (a-f):

$$\text{SameTypeEquals}(l,r) := \begin{cases} \top & \text{Type}(l) = \tau_{\text{undefined}} \\ \top & \text{IsNull}(l) \\ \text{NumberEquals}(\text{ToNumber}(l), \text{ToNumber}(r)) & \text{Type}(l) = \tau_{\text{number}} \\ \text{ToString}(l) = \text{ToString}(r) & \text{Type}(l) = \tau_{\text{string}} \\ \text{ToBoolean}(l) = \text{ToBoolean}(r) & \text{Type}(l) = \tau_{\text{boolean}} \\ \text{ToObject}(l) = \text{ToObject}(r) & \text{else} \end{cases}$$

NumberEquals handles case 1c (i-vi):

$$\text{NumberEquals}(l,r) := \begin{cases} \bot & \text{isNaN}(l) \\ \bot & \text{isNaN}(r) \\ \top & l = r \\ \top & l = +0 \wedge r = -0 \\ \top & l = -0 \wedge r = +0 \\ \bot & \text{else} \end{cases}$$

5.6.8.2 The Does-not-equals Operator (`!=`)

Type is τ_{boolean}. Value of `l != r` is $\neg\,\text{Equals}(l,r)$ (see Section 5.6.8.1).

5.6.8.3 The Strict Equals Operator (`===`)

Type is τ_{boolean}. Value of `l === r` is $\text{StrictEquals}(l,r)$. The encoding is similar to *The Strict Equality Comparison Algorithm* [ES5, Sec. 11.9.6].

$$\text{StrictEquals}(l,r) := (\text{Type}(l) = \text{Type}(r)) \wedge e$$

$$e := \begin{cases} \text{ToBoolean}(l) = \text{ToBoolean}(r) & \text{Type}(l) = \tau_{\text{boolean}} \\ \text{ToFunction}(l) = \text{ToFunction}(r) & \text{Type}(l) = \tau_{\text{function}} \\ n \wedge \text{ToNumber}(l) = \text{ToNumber}(r) & \text{Type}(l) = \tau_{\text{number}} \\ \text{ToObject}(l) = \text{ToObject}(r) & \text{Type}(l) = \tau_{\text{object}} \\ \text{ToString}(l) = \text{ToString}(r) & \text{Type}(l) = \tau_{\text{string}} \\ \text{Type}(l) = \tau_{\text{undefined}} & \text{else} \end{cases}$$

$$n := \neg\,\text{isNaN}(\text{ToNumber}(l)) \wedge \neg\,\text{isNaN}(\text{ToNumber}(r))$$

5.6.8.4 The Strict Does-not-equal Operator (`!==`)

Type is τ_{boolean}. Value of `l !== r` is $\neg\,\text{StrictEquals}(l, r)$ (see Section 5.6.8.3).

5.6.9 Binary Bitwise Operators

Type is τ_{number}. Operands are casted to bit-vector formulas using ToInt32 to be able to use the respective operator of bit-vector theory. The result is casted back to a floating-point number (see Section 5.2.4) and represents the value.

5.6.10 declaredBy

A `declaredBy` expression `id declaredBy functionDeclaration` (see Section 4.2.2.1) has an identifier `id` and a function declaration `functionDeclaration` as operands. The type is τ_{boolean}. The value is

$$\text{declarationOf}(\text{functionValue}(v)) = d$$

where v is the value of the encoded variable `id` as described in Section 5.6.1.2 and d is the declaration-ID of `functionDeclaration` (see Section 5.7.3).

5.7 Operations

This section describes the formula encoding of operations that are part of the CFA (see Section 4.2.1). Each encoding of an operation is described as a boolean formula. This formula might be described by the logical conjunction of a set of conditions that we call *constraints*. All constraints that are mentioned in the description of the operation and its expressions are part of that set. After all, we define the strongest post operator of a program path as the logical conjunction of the boolean formulas of all path operations.

5.7.1 Assumption

An assumption [`p`], is encoded as ToBoolean(p).

5.7.2 Variable Declaration

A variable declaration `var x` is handled like an assignment operation (see Section 5.7.4) `x = undefined`. A variable declaration `var x = e` is handled like an assignment operation `x = e`.

Note

Strictly speaking, variable declaration operations could be replaced by their respective assignment operations in the CFA. However, in my opinion it is easier to understand CFA examples and transformation rules if they are kept.

5.7.3 Function Declaration

A function declaration operation `function func(args*) { ... }` indicates that the function object of `func` is created. The object is created similar to an Object Initialiser expression (see Section 5.6.1.5)

```
{
    prototype: {},
    length: len
}
```

where `len` represents the count of function parameters. The object-ID o of the created object is then used in the final constraints

$$\begin{aligned} \text{typeof}(fv) &= \tau_{\text{function}} \\ \text{functionValue}(fv) &= o \\ \text{objectValue}(fv) &= o \\ \text{scopeOf}(o) &= \mathit{currentScope} \\ \text{declarationOf}(o) &= d \end{aligned}$$

where fv is the scoped variable of the function declaration identifier `func` and d is the declaration-ID of the declared function.

Note

objectValue(fv) = o is done to allow the direct access (without type check) to objectValue(o) if o is expected to be a (function) object (for example in a property accessor expression `o.p`).

A unique declaration-ID is used for every function declaration. It is used in `declaredBy` expressions (see Section 5.6.10), which are used to resolve unknown function calls (see Section 4.3.1.12) and unknown constructor calls (see Section 4.3.1.13).

Note

All function instances of a function declaration have the same declaration-ID, but different object-IDs.

5.7.4 Assignment

An assignment `lhs = e` assigns an expression `e` to a left hand side `lhs` that is either a property-access operator expression or an identifier reference. These cases have to be handled differently. If `e` is a function return variable (assignment operation of function exit edge), Section 5.7.4.3 has to be considered.

5.7.4.1 Assignment To Identifier

This section describes the case that the left hand side of an assignment `x = e` is an identifier reference `x`. First, make the formula e of `e`, since it may contain a reference to `x`. Then, the index counter of variable x is increased from its *old* index to its *new* index. As explained in Section 5.3.2.1, other scoped variables (same declaration, but different scope-ID) have to be updated. Therefore, all scope-IDs that are created for a function declaration (on a call) are added to its set of scope-IDs (see Section 5.7.6). For each scope-ID s in this set a constraint

$$sx = s \vee \text{var}(s, x_{old}) = \text{var}(s, x_{new})$$

is added where sx is the scope of the declaration of `x` (see Section 5.3.2). Finally, the actual assignment is encoded using $\text{Assignment}(\text{var}(sx, x_{new}), e)$.

$$\text{Assignment}(l, r) := \text{Type}(l) = \text{Type}(r) \wedge a$$

where l is a variable-ID, r is an expression and

$$a := \begin{cases} \text{booleanValue}(l) = \text{ToBoolean}(r) & \text{Type}(r) = \tau_{\text{boolean}} \\ \text{functionValue}(l) = \text{ToFunction}(r) \wedge \text{objectValue}(l) = \text{ToObject}(r) & \text{Type}(r) = \tau_{\text{function}} \\ \text{numberValue}(l) =_{\text{assign}} \text{ToNumber}(r) & \text{Type}(r) = \tau_{\text{number}} \\ \text{stringValue}(l) = \text{ToString}(r) & \text{Type}(r) = \tau_{\text{string}} \\ \text{objectValue}(l) = \text{ToObject}(r) & \text{Type}(r) = \tau_{\text{object}} \\ \text{Type}(l) = \tau_{\text{undefined}} & \text{else} \end{cases}$$

Note

$\text{objectValue}(l) = \text{ToObject}(r)$ is also done in case of $\text{Type}(r) = \tau_{\text{function}}$ to allow the direct access (without type check) to $\text{objectValue}(o)$ if o is expected to be a (function) object (for example in a property accessor expression `o.p`).

5.7.4.2 Assignment To Object Property

This section describes the case that the left hand side of the assignment `lhs = e` is a property-access operator expression. First, make the formula e of `e`, since it may contain a reference to the object or the property of the left hand side.

5.7.4.2.1 Dot Notation In case `lhs` is a property accessor in dot notation `obj.propName`, it consists of an expression `obj` and an identifier name `propName`. The identifier name is used as string value (here StringID(`"propName"`)) of the property.

Create a fresh variable-ID p and mark it as set property (variable) using the constraint

$$p \neq \mathit{objectFieldNotSet}$$

and assign the value using $\text{Assignment}(p, e)$ from Section 5.7.4.1. Finally, update the fields of the object using a constraint:

$$\text{SetObjectFields}(o, \text{store}(\text{GetObjectFields}(o), \text{StringID}(\texttt{"propName"}), p))$$

where $o := \text{ToObject}(obj)$ and

$$\text{SetObjectFields}(o, ps) := \mathit{objectFields}_{new} = \text{store}(\mathit{objectFields}_{old}, o, ps)$$

where o is an object-ID, ps is an array of properties, and the index counter of variable *objectFields* is increased from its *old* index to its *new* index.

Note

GetObjectFields is defined in Section 5.6.2.1.1.

5.7.4.2.2 Bracket Notation

In case `lhs` is a property accessor in bracket notation `obj[propExpr]`, it consists of expressions `obj` and `propExpr`. It is handled similar to an assignment to the property accessor in dot notation (see Section 5.7.4.2.1) with ToString(*propExpr*) as property name. However, in contrast to a property accessor in dot notation, the accessed property of a property accessor in bracket notation may be an array index.

Note

Dot notation has an identifier as property name, which can not be an array index (string), since identifiers must not begin with a digit [ES5, Sec. 7.6].

Hence, the length property (if object is an array) might be changed, too. This is encoded as a constraint

$$\text{UpdateArrayLengthProperty}(o, \text{GetObjectFields}(o), \text{ToString}(propExpr))$$

where $o := \text{ToObject}(obj)$ and UpdateArrayLengthProperty is defined as

$$\text{UpdateArrayLengthProperty}(o, ps, p) := \text{store}(ps, \text{StringID}(\texttt{"length"}), l)$$

where o is an object-ID, ps is the array of properties of o, p is the property name (string-ID), and

$$\begin{aligned} l &:= \begin{cases} length_{\text{new}} & \text{isArray}(o) \wedge \text{IsArrayIndexString}(p) \wedge oldLength < newLength \\ length_{\text{old}} & \text{else} \end{cases} \\ oldLength &:= \text{numberValue}(length_{\text{old}}) \\ newLength &:= \text{StringToNumber}(p) + 1 \\ length_{\text{old}} &:= \text{AccessField}(o, \text{StringID}(\texttt{"length"})) \end{aligned}$$

where $length_{\text{old}}$ represents the old property variable of the *length* property and $length_{\text{new}}$ is a fresh variable-ID (new property variable of the *length* property if updated) that is marked as a set property variable using the constraint

$$length_{\text{new}} \neq objectFieldNotSet$$

and associated with its value using the constraint

$$\text{typeof}(length_{\text{new}}) = \tau_{\text{number}} \wedge \text{numberValue}(length_{\text{new}}) =_{\text{assign}} newLength$$

Note

The *length* property is only updated if the object o is an array, the property p is an array index (string), and an element after the current last element is assigned. In that case, the new length value is the index (number value of p) increased by one (index counting starts from 0)

5.7.4.3 Assignment Of Return Variable

An assignment operation `lhs = r` of a function exit edge has to be handled specially. It assigns a return variable `r` to a left hand side `lhs`. On the one hand side, the execution context [ES5, Sec. 10.4] is switched back from the *called* function to the *caller* (function or global code). That means, the *currentScope* of the called function has to be used when creating the formula of the

(variable) identifier reference `r` (see Section 5.6.1.2) and the *currentScope* of the caller has to be used when creating the formula of `lhs`. On the other hand, the function exit edge might be associated with a constructor call operation. In that case, the type of the return variable has to be checked[7]. If the type equals the function or object type and the objectValue is not *null*, do the regular assignment of `lhs = r`. Otherwise, handle it like `lhs = this` where *this* is the this variable of the called constructor.

5.7.5 Delete Operation

A delete operation is equivalent to assigning *objectFieldNotSet* to the property of the object (not to the prototype) that is deleted (see Section 5.7.4.2).

5.7.6 Function Call

Thanks to the preprocessing, the function declaration of each function call is known. Thus, the parameter variables, the this-variable, and function object argument variable of the (called) function declaration are known. On a function call the execution context [ES5, Sec. 10.4] switches from the *caller* (function or global code) to the *called* function. The following has to be done:

- create a new scope for the called function
- update current scope stack
- bind (optional) this argument
- assign arguments of call to parameter variables of called function

Create a new (unique and unused) scope-ID s and add it to the set of scope-IDs of the called function. Let *calledScopeVariable* be the current scope variable of the called function. Increase the static index of *calledScopeVariable* to *new* and assign s to it using the constraint $\mathit{calledScopeVariable}_{new} = s$. Associate the scope s with its scope stack using the constraint

$$\text{scopeStack}(s) = \text{store}(ss, n+1, s)$$

where n is the nesting level of the called function declaration

> **Note**
>
> The scope s is itself part of its scope stack. It is on top of the stack at index $n+1$, which is equal to the nesting level of local declarations in the called function. If an identifier of these declarations is referenced from a deeper nested function declaration, then the scope s will be selected.

and ss is a part of the scope stack of the caller that is defined as

$$ss := \begin{cases} \mathit{globalScopeStack} & \text{declaration of called function is global} \\ \text{scopeStack}(\text{scopeOf}(\text{functionValue}(d))) & \text{function object argument exists} \\ \text{scopeStack}(\text{select}(\text{scopeStack}(\mathit{callerScopeVariable}), n)) & \text{else} \end{cases}$$

where d is the scoped variable of the called function and *callerScopeVariable* is the current scope variable of the caller.

[7] The return value of a constructor is usually the created object (this variable). However, the return value can be overwritten by a return statement that returns an object. [ES5, Sec. 13.2.2]

Note

The scope stack of a global function is globally known as *globalScopeStack*. Hence, it is not required to select it from the scope stack of the caller. In case of a function instance, the scope stack of the function instance has been associated with the scope of the function instance by a function declaration operation (see Section 5.7.3). Else, the scope stack is selected from the scope stack of the caller, whereas the index is the nesting level of the declaration. The selected scope stack has been put on top by a call of the surrounding function at nesting level n. However, the (statically indexed) *currentScope* variable of this function can not be used, since it might have been changed due to another call to the same function declaration.

Lastly, the function arguments are passed by assigning (see Section 5.7.4) them to the parameters that are handled like scoped variables of the called function. Likewise, the `this` argument is assigned to the this variable (see Section 5.6.1.1) of the called function declaration as well as the function object argument.

5.7.7 Constructor Call

A constructor call `new func(e*)` is handled like a function call operation `func(e*)` (see Section 5.7.6), but instead of the this argument a new object is created and assigned to the this variable. This object is created similar to the Object Initialiser (see Section 5.6.1.5)

```
{
    [[Prototype]]: func.prototype
}
```

where `[[Prototype]]` represents the prototype property of Section 5.2.6.2.

6 Implementation

The implementation can be found in the CPAchecker repository[1]. It includes the ECMAScript parser-frontend that creates a program representation (CFA) as described in Chapter 4 and the formula encoding of the ECMAScript operations and the language itself as described in Chapter 5.

CPAchecker is written in Java. The ECMAScript parser-frontend[2] uses the parser of the Eclipse JavaScript Development Tools[3]. The parser creates an *abstract syntax tree* (AST) from the ECMAScript source code. It is then transformed considering the preprocessing of Section 4.3 to an internal representation[4] (another AST) that is independent from the used parser[5] and that is used to model the CFA operations desribed in Chapter 4.

CPAchecker contains abstract classes and interfaces for AST nodes that (supported) languages have in common. Thereby, most of the core logic (algorithms) can be shared and only needs (language specific) adjustments where necessary. Hence, only the formula encoding of Chapter 5 has to be implemented to be able to use the SMT based approaches of CPAchecker. It is added in the package `org.sosy_lab.cpachecker.util.predicates.pathformula.jstoformula`.

The implementation only contains a specification that defines a call of a function `__VERIFIER_error` as error location. However, CPAchecker allows more complex specifications using monitor automata [4].

6.1 Configuration Options

CPAchecker offers several configuration options. We add further options that influence the ECMAScript specific SMT formula encoding and outline there impact.

6.1.1 Maximum Field Count

As described in Section 5.2.6.1, object properties are managed as an array formula that maps each property name (string-ID formula) to a variable formula that represents the value of the property. If a property is not set on an object, then the property name is mapped to a special variable formula *objectFieldNotSet*, which represents an unset field. *emptyObjectFields* represents the (initial) property mapping, where all property names are mapped to *objectFieldNotSet*. Therefore, we could use an all-quantifier, but the combination of uninterpreted functions (used in the formula encoding) and quantifiers is undecidable [7]. Hence, we explicitly store *objectFieldNotSet* for all string-IDs of non-number strings[6] used in the program instead. It would be too much values to do this for all possible floating point formulas (string-IDs). We already left non-number strings. Thereby, we have a lower bound for the range of string-IDs of non-number strings. It would be still too much (unnecessary) values above this bound to cover. Hence, the upper bound of string-IDs is defined by a configuration option.

[1] `https://svn.sosy-lab.org/software/cpachecker/branches/javascript/`

[2] See package `org.sosy_lab.cpachecker.cfa.parser.eclipse.js`.

[3] `https://www.eclipse.org/webtools/jsdt/core/`

[4] See package `org.sosy_lab.cpachecker.cfa.ast.js`.

[5] This makes the parser exchangeable.

[6] All strings that are the result of applying ToString to the Number type [ES5, Sec. 9.8.1] are called *number strings*.

It can be detected if there exist more known string-IDs[7] in the analyzed program, but unknown string-IDs[8] remain. It would be possible to set the upper bound to the count of known string-IDs, but this has not been implemented yet. Instead the user is informed if the upper bound defined is lower than the count of known string-IDs.

The drawback of this approach is that it is unknown if a property with a name that has an unknown string-ID is unset before it is set the first time. However, this is a very unusual case that occurs only in very few programs.

6.1.2 Maximum Prototype Chain Length

As described in Section 5.2.6.3, we assume that no prototype chain in the analyzed program is longer as a maximum *maxPrototypeChainLength*. This value is defined as a special option. If this value is too small, properties higher up in the prototype chain are not taken into account. This might lead to false results. However, in most cases it should be possible to estimate an adequate value. In case of doubt, the value can be increased, but larger values might impact the performance.

6.1.3 Usage Of NaN and infinity

CPAchecker provides options to encode floating point formulas as rational formulas. This can be useful if a solver does not support floating point theory. Moreover, rational formulas are less expensive to solve.

However, rational formulas do not support the values *NaN* and $\pm\infty$, but only as a variable and not as a value. Checking for *NaN* or $\pm\infty$ can lead to satisfiable and non-tautological formulas. Hence, an option is added that alters the formula encoding by assuming that those checks always result in $\bot$. For example, NumberToString of Section 5.5.3 is defined as:

$$\text{NumberToString}(n) := \begin{cases} \text{StringID}(\texttt{"NaN"}) & \text{isNaN}(n) \\ \text{StringID}(\texttt{"-Infinity"}) & n = -\infty \\ \text{StringID}(\texttt{"Infinity"}) & n = \infty \\ \text{cast } n \text{ to floating point type of string formula} & \text{else} \end{cases}$$

It is reduced to the *else* case since all other cases are assumed to be $\bot$.

6.2 Unimplemented Features

Due to lack of time, the `==`, `!=`, `in` and `instanceof` operators have not been implemented yet. `===` is used instead of `==`. `!==` is used instead of `!=`. Further, the remainder operator of the floating point theory is not part of the unified Java API for SMT solvers[9] used by CPAchecker. It is currently only supported by the solver Z3[10]. In the meantime, we use a formula encoding similar to the ECMAScript specification of the remainder [ES5, Sec. 11.5.3] that at least covers all cases, where either infinity, zero, or *NaN* is involved.

Besides, `callUnknownFunction` and `callUnknownConstructor` have not been inlined yet and are called like regular functions. Thereby, recursive calls of `callUnknownFunction` or `callUnknownConstructor` occur in case of a nested dynamic function call. For example, if an unknown function `f` is

[7] String-IDs of string constants and property names.
[8] String-IDs of number strings and the result of string operations (concatenation) that do not exist as string constant in the program.
[9] `https://github.com/sosy-lab/java-smt`
[10] `https://github.com/Z3Prover/z3`

called, it results in a call `callUnknownFunction(f)` that results in a call of a known function. If this known function contains another call to an unknown function `g`, it results in another call `callUnknownFunction(g)`, which is an indirect recursive function call of `callUnknownFunction`.

Note

The function declarations of `callUnknownFunction` and `callUnknownConstructor` in our implementation use the maximum parameter count of all function declarations plus one (because of the additional parameter `functionObject`). If there exist two function declarations `f(p0)` and `g(p0, p1)`, then the maximum parameter count is 2, which results in the declarations `callUnknownFunction(functionObject, p0, p1)` as shown in Listing 4.2 and `callUnknownConstructor(functionObject, p0, p1)` as shown in Listing 4.5.

As we see in Chapter 7, some files can not be analyzed due to these issues.

7 Evaluation

In this chapter we will look at the evaluation of the functional correctness of the implementation of the formula encoding based on the test programs of the official ECMAScript Conformance Test Suite *Test262*[1] using bounded model checking [5] with k-induction [12, 29].

The goal of Test262 is to provide test material that covers every observable behavior specified in the ECMA-414 Standards Suite[2]. The development of test262 is an on-going process. Tests have been contributed by ECMA members, browser vendors (Microsoft and Google), TC39 member organizations and members of the world-wide ECMAScript community. As of May 2019, Test262 consisted of nearly 31000 individual test files. Each of these files contains one or more distinct test cases. This marks the most comprehensive ECMAScript test suite to date. TC39 does not consider the coverage to be complete, but the test coverage of ECMAScript 5.1 is broad and test files do not contain more features than necessary. Overall, it is well suited for our evaluation. We use the latest version[3] of Test262 (May 3rd 2019) in this evaluation.

However, we have to exclude some files, since Test262 also contains files with features that are not supported (see Section 4.1) or not implemented yet (see Section 6.2). Therefore, we initially benefit from the structure of the test suite. Files with newer and unsupported features can easily identified by directory and file names. However, this is not sufficient to exclude all files. Fortunately, files contain metadata in a comment. We exclude all files that are labelled to only check syntax errors. Besides, the metadata contains a list of flags and used features that we use to exclude respective files. Not all files contain a complete feature list in the metadata, but some files contain a key *es6id* in the metadata. We exclude them, but the key *es6id* is deprecated[4] in favor of the key *esid* and not used in all files with ES6 features. *esid* provides no direct information about the assumed ECMAScript version. Hence, we additionally parse[5] the remaining files[6] and check if certain AST nodes are present that indicate unsupported features. Besides, we exclude files with elided array elements, since they are not parsed correctly by the Eclipse parser (used in the implementation) due to a known bug[7]. We end up with 780 files that we try to verify.

Normally, these tests would be executed (independently from each other), whereas two files that provide several assertion functions would be evaluated before the tested code. These assertion functions throw a special exception that indicates a failure. Since we do not cover exceptions yet and the implementation only contains a specification that defines a call of a function `__VERIFIER_error` as error location, we use a simple workaround. We define similar assertion functions that call `__VERIFIER_error` instead of throwing an error.

In addition to that, we evaluate the following file that acts as a polyfill[8] for a small amount of built-ins in order to cover more tests:

```
function isNaN(value) {
  return value === undefined || value !== value;
}
```

[1] https://github.com/tc39/test262
[2] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-414.pdf
[3] https://github.com/tc39/test262/tree/d47749e84daeea28b6fa7cefd69e7f2836dbbf37
[4] https://github.com/tc39/test262/wiki/Test262-Technical-Rationale-Report,-October-2017
[5] Using the parser https://github.com/Kronuz/esprima-python
[6] We need the previous filter criteria, since the parser would not be able to parse all files due to certain issues or unsupported features. All remaining files are parsable.
[7] https://bugs.eclipse.org/bugs/show_bug.cgi?id=544733
[8] A file that reimplements built-in functions using existing ECMAScript code.

```
// constructor call not supported yet
function Number(value) {
  return +value;
}

// constructor call not supported yet
function Boolean(value) {
  return !!value;
}

Number.NaN = NaN;
Number.POSITIVE_INFINITY = Infinity;
Number.NEGATIVE_INFINITY = -Infinity;
Number.MAX_VALUE = 1.7976931348623157E308;
Number.MIN_VALUE = 4.9E-324;

// constructor call not supported yet
function Date() {
  // dummy value
  return "Mon Feb 25 2019 12:06:04 GMT+0100 (Central European Standard Time)";
}
```

As mentioned in the beginning of this chapter, we choose bounded model checking (BMC) with k-induction to analyze the test files. Thereby, we can unroll loops and avoid interpolation of iteration variables. ECMAScript only has a number type and we encoded number values as floating point formulas. Solvers only have bad interpolation support of floating point formulas. Hence, we could not analyze many files that contain loops.

Before we can run CPAchecker, we need to configure it. For example, we need to specify the maximum of loop iterations k. We choose k to be 11. As it turns out, this value is large enough to cover all tested loops[9]. We initially choose a maximum field count (see Section 6.1.1) of 80 and a maximum prototype chain length (see Section 6.1.2) of 5. As we will see later, the maximum field count is not big enough for all tested files. Hence, we will increase it for those. Apart from that, we use the default solver of CPAchecker known as MathSAT5[10].

We run all tests as a benchmark in the VerifierCloud[11] using BenchExec [32] and the CPAchecker version of our implementation[12] on machines with Intel Xeon E3-1230 v5 (3.40 GHz) CPU using 4 cores, with a time limit of 15 minutes, a memory limit of 15 GB, and a heap limit of 13000MiB for Java. We do multiple runs as described in the following. Table 7.1 gives an overview of the results of each run.

Run	Description	Files	Correct	Incorrect	Unknown
1	Positive tests after automatic filtering	780	641	42	97
2	Positive tests after manual filtering and reconfiguring failed tests of 1st run	664	662	0	2
3	Negative tests of correct tests of 2nd run	8625	8593	13	19

Table 7.1: Results of different evaluation runs

In the first run[13], we test all 780 files that are left after the filtering based on the criteria described above. 641 files pass, but 139 files fail. 42 files are incorrect, whereof 9 files access

[9] CPAchecker provides options to adjust (increase) this value automatically if a proper value is unknown.
[10] http://mathsat.fbk.eu/
[11] `https://vcloud.sosy-lab.org`
[12] `https://svn.sosy-lab.org/software/cpachecker/branches/javascript/?p=31149`
[13] Benchmark configuration file of first run `https://svn.sosy-lab.org/software/cpachecker/branches/javascript/test/test-sets/bmc-JavaScript-test262-benchmark-1.xml?p=31149`

the global object, 25 files use unsupported string operations, 7 files rely on non strict equal semantic[14], and 1 file fails because of improperly handled unicode in the function name. 97 files are unknown[15], whereof 1 file accesses the global object, 6 files use unsupported string operations, 35 files contain special characters (whitespace, unicode, etc.) that are not handled properly, 9 files have nested dynamic function calls[16], 7 files contain recursive function calls, 8 files are parsed wrong by the Eclipse parser, 1 file contains a with-statement, 7 files contain unimplemented remainder operations, 11 files that led to a timeout, and 12 files that require a greater maximum field count.

In the second run[17], we exclude the files that contain unsupported or unimplemented features. We change the run configuration for the remaining 23 unknown files of the previous run. Of the 11 files that led to a timeout, we try to run all with float encoded as rational while considering the adjustments of the formula encoding described in Section 6.1.3. Further, we decrease the maximum of loop iterations k to 2, where it is sufficient (3 of the 11 files). Apart from this, 12 unknown files require a greater maximum field count. They are all more than 2500 lines long and assert multiple shift operations (more than 500 assertions). We set the maximum field count to a generous value of 1000. Thanks to these configuration changes, 21 of the 23 unknown files pass. 2 files still timeout. Both contain multiple loops. One uses floating point operations (including division) to change the iteration variables of loops. The other contains multiple tested loops. Nevertheless, we added 21 files to the count of passed files 641 of the first run adding up to 662 passing files up to this point.

So far, we only tested the positive cases. That means that we checked that a test passes as expected. However, it has already been shown during development that errors can remain hidden. Once, a bug was introduced in the formula encoding, whereby no location after a specific kind of statement was reachable. The consequence of this was that conditions were not checked (reached) that would have lead to a failure. Hence, all tests magically passed. To compensate this issue, we generated *negated tests* for each test file that was correct in the second run and check that these tests fail.

Note

Thanks to this approach we found a bug[a] in two test files of Test262. This bug has been fixed in the version of Test262 that we use in our evaluation.

[a] `https://github.com/tc39/test262/issues/2049`

There are basically two forms of assertions that are used. On the one hand, there are assertion functions like `assert(condition, message)`, where the condition is checked inside. Such calls can be negated by negating the condition inside of the function definition. Therefore, we only need to evaluate another file for the assertion functions. On the other hand, there is an assertion function `$ERROR(message)` that always fails when called. In almost all cases, those calls look like this:

```
if (condition) { $ERROR(...) }
```

We generate a negated test file for each of these occurrences and negate the condition of the if-statement like this:

```
if (!(condition)) { $ERROR(...) }
```

Thereby, we get 8625 negated tested files in total.

[14] Non strict equal operators are handled like strict operators in the implementation.

[15] Solver could not solve formula due to a timeout, out of memory, etc.

[16] Causes recursive function call of `callUnknownFunction` or `callUnknownConstructor` that have not been inlined yet in the implementation.

[17] Benchmark configuration file of second run `https://svn.sosy-lab.org/software/cpachecker/branches/javascript/test/test-sets/bmc-JavaScript-test262-benchmark-2.xml?p=31149`

We run all negated tested files with same configuration of the second run. In this third run[18], 8593 files pass, but 32 files fail. 13 files are incorrect, whereof 1 file accesses the global object and the others belong to files, where the negated condition check is not reachable. These files contained an assert call of the form

```
if(false)
  if (true)
    $ERROR('message');
```

where we negate the condition of the surrounding if-statement like this

```
if(false)
  if (!(true))
    $ERROR('message');
```

We expected that this negated test fails, but negated condition check is not reachable. Thus, the negated test passes, which is what we actually should have expected. The other 19 of 32 failed files are unknown, whereof they were generated from 4 files. One of those files contains an unsupported remainder operation. The others timeout due to unknown reasons.

After running the negated tests, we know that 2 files contain unsupported features, even though their positive tests passed. We should not count them as covered tests to be fair. That means we subtract them from the correct file count 662 of the second run and end up with a total 660 of covered tests, whereas we do not know if the counter-checks (negated tests) of 4 files would have all passed.

The evaluation also revealed a few bugs in our implementation caused by improper handled special characters. 21 files required to encode float as rational. Otherwise, they let to a timeout. In addition, 2 files let to a timeout even though we adjusted the configuration. In that regard, it should be noted that the implementation is not really optimized yet. For example, the performance may be improved in future work as described in Section 9.8.

[18] Benchmark configuration file of third run `https://svn.sosy-lab.org/software/cpachecker/branches/javascript/test/test-sets/bmc-JavaScript-test262-benchmark-3.xml?p=31149`

8 Conclusion

In this paper, we extended CPAchecker to a restricted subset of ECMAScript 5.1 by adding a respective parser frontend and an operator that is responsible for encoding the semantics of program operations into SMT formulas.

We dealt in Chapter 5 with the challenges mentioned in Chapter 2. The ECMAScript standard uses internal functions to describe the semantics of statements and expressions. Section 5.5 showed with type conversion functions the closest encoding (with regard to the description in the standard) of internal functions, but also the object property management is described using internal functions. Section 5.2.6 outlined how objects and properties are formula encoded and how the lookup on the prototype chain is done in general. Later, we saw in Section 5.6.2.1, Section 5.6.7.5, and Section 5.6.7.6 how the lookup works in practice. Therewith, we also reasoned about extensible objects and dynamic property access. We did not reason about property descriptors and property traversal (see Section 4.1). We will discuss in Chapter 9 how we may reason about this in future work. Apart from that, we reasoned about higher-order functions that required calls to unknown functions (see Section 4.3.1.12 and Section 4.3.1.13) that we resolved using a special operator `declaredBy` (see Section 4.2.2.1). Further, we described in Section 5.3.2 how we reason about scope chains and function closures of arbitrary complexity.

We evaluated the functional correctness of the implementation described in Chapter 6 based on the official ECMAScript Conformance Test Suite *Test262* in Chapter 7 considering the assumptions in Section 4.1 and the unimplemented features as mentioned in Section 6.2. The current implementation covers 660 test files. This is significantly less than the test coverage of KJS (2782 files) and JaVerT (8797 files). The main reason are missing features (exceptions, built-ins, etc.). However, we provided a basis for future work to cover more tests.

9 Future Work

Last but not least, we would like to give an outlook on how our work can be expanded and improved.

9.1 The for-in Statement

This is one of the most challenging features. The for-in Statement iterates only over enumerable properties on the entire prototype chain. Further, the mechanics and order of enumerating the properties are not specified. It might be a start to look at the work of Cox, Chang, and Rival[9], who have shown how to reason about property iteration in a simple extensible object calculus. Another idea would be to unroll the loop by checking all properties considering the maximum field count (see Section 6.1.1) and using the in-operator described in Section 5.6.7.6.

9.2 The with Statement

It should be possible to encode the with statement [ES5, Sec. 12.10] in the CFA using the existing operations in Section 4.2.1. This has not been done in this work, since the with statement is deprecated. However, it has to be considered to get fully ECMAScript 5.1 compliant.

9.3 Exceptions

The difficulty in handling exceptions is modeling the catching of exceptions over multiple functions. One way to model this is to save the catch block nodes across the CFA. If an exception can be thrown in a statement, this would be expressed by an assume-edge to the exception node. If an exception is thrown beyond the CFA of the global code, it would point to a node with an outgoing edge. This edge would be labelled as *uncaught Exception*; and points to a node with no outgoing edges (program exit node).

9.4 Standard Built-in ECMAScript Objects

Several built-ins are listed in the ECMAScript standard [ES5, Sec. 15]. `Object`, `Function`, `Array`, `String`, `Boolean`, `Number`, `Math`, and `Error` could be encoded similar to a regular object in the most general sense (see Section 5.2.6). Some properties and methods as well as functions of the global object might require a specialized formula encoding to describe the behaviour, but most of them should be straight forward. On the other hand, `Date`, `RegExp`, and `JSON` require further research including a more powerful string encoding to cover their main purpose.

Apart from that, we did not consider (see Section 4.1) that global variables are actually stored in the global object. We could encode the global object as a regular object to store global variables in it, but another approach might be imaginable.

9.5 Property Descriptors

As described in Section 5.2.6.1, properties are managed as an array formula that maps each property name (string-ID formula) to a variable formula that represents the value of the prop-

erty. Instead of mapping the property name directly to the value representation, it could be mapped to an encoding of its property descriptor that includes a reference to the variable formula that represents the value of the property. The encoding of the property descriptor should be straight forward as the possible attributes are known. As a consequence, the formula encoding of operations such as the property access have to be adjusted depending on the formula encoding of the property descriptor taking respective attributes into account.

9.6 Implicit Function Calls From Internal Methods

Implicit function calls from internal methods[1] may require to add assumption edges to the CFA that represent the checks used in internal methods that might lead to a call of a regular (not internal) function. The exact realization of each internal method requires further research.

9.7 arguments

The Arguments Object [ES5, Sec. 10.6] `arguments` can be created on a function call similar to a regular object as described in Section 5.2.6 and passed in a similar way as the `this` object as described in Section 5.7.6.

9.8 Performance Improvements

The performance of the implementation may be improved in several ways. One idea is to reduce the amount of strings that are used as property names in the approach described in Section 6.1.1 using heuristics. For example, it is unlikely that very long strings or strings with many special characters will be used as property names. Apart from that, only captured local variables have to be encoded as scoped variables as described in Section 5.3.2. All other variables can be simply encoded as statically indexed variables as described in Section 5.3.1. Finally, CPAchecker allows to combine different CPAs. A CPA that tracks the type of variables and expressions may allow to simplify SMT formulas. This may allow to encode numbers as integers instead of floating point formulas in some cases (for example, iteration variables of loops are usually used as integers).

9.9 Specification

Our implementation only contains a specification that defines a call of a function `__VERIFIER_error` as error location. CPAchecker supports the matching of expression and statement in its specification definition for the language C. A similar implementation can be implemented for ECMAScript to overcome the current restriction.

9.10 Maximum Field Count

The option for the maximum field count as described in Section 6.1.1 can be automatically determined based on the known strings in the code. At the moment, the implementation only notifies the user that the used value is to small if more strings appear.

[1] For example, `valueOf` might be called by the internal method *[[DefaultValue]]* [ES5, Sec. 8.12.8].

9.11 Maximum Prototype Chain Length

Heuristics might be added to determine an appropriate maximum prototype chain length (see Section 6.1.2). The details require further research.

List of Figures

Listings

Bibliography

[1] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. "Towards Type Inference for JavaScript". In: *ECOOP 2005 - Object-Oriented Programming*. Ed. by Andrew P. Black. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 428–452. ISBN: 978-3-540-31725-8.

[2] Esben Andreasen and Anders Møller. "Determinacy in Static Analysis for jQuery". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 17–31. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660214. URL: http://doi.acm.org/10.1145/2660193.2660214.

[3] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. "A Decision Procedure for Bit-vector Arithmetic". In: *Proceedings of the 35th Annual Design Automation Conference*. DAC '98. San Francisco, California, USA: ACM, 1998, pp. 522–527. ISBN: 0-89791-964-5. DOI: 10.1145/277044.277186. URL: http://doi.acm.org/10.1145/277044.277186.

[4] Dirk Beyer et al. "The Blast Query Language for Software Verification". In: *Static Analysis*. Ed. by Roberto Giacobazzi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 2–18. ISBN: 978-3-540-27864-1.

[5] Armin Biere et al. "Symbolic Model Checking without BDDs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-49059-3.

[6] Gavin Bierman, Martín Abadi, and Mads Torgersen. "Understanding TypeScript". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN: 978-3-662-44202-9.

[7] Egon Börger, Erich Graedel, and Yuri Gurevich. *The Classical Decision Problem*. Jan. 1997. DOI: 10.1007/978-3-642-59207-2.

[8] A. Brillout, D. Kroening, and T. Wahl. "Mixed abstractions for floating-point arithmetic". In: *2009 Formal Methods in Computer-Aided Design*. Nov. 2009, pp. 69–76. DOI: 10.1109/FMCAD.2009.5351141.

[9] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. "Automatic Analysis of Open Objects in Dynamic Language Programs". In: *Static Analysis*. Ed. by Markus Müller-Olm and Helmut Seidl. Cham: Springer International Publishing, 2014, pp. 134–150. ISBN: 978-3-319-10936-7.

[10] David Cyrluk, Oliver Möller, and Harald Rueß. "An efficient decision procedure for the theory of fixed-sized bit-vectors". In: *Computer Aided Verification*. Ed. by Orna Grumberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 60–71. ISBN: 978-3-540-69195-2.

[11] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320.

[12] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. "Automatic analysis of DMA races using model checking and k-induction". In: *Formal Methods in System Design* 39.1 (Aug. 2011), pp. 83–113. ISSN: 1572-8102. DOI: 10.1007/s10703-011-0124-2. URL: https://doi.org/10.1007/s10703-011-0124-2.

[ES5] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1. June 2011. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[13] Asger Feldthaus and Anders Møller. "Checking Correctness of TypeScript Interfaces for JavaScript Libraries". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 1–16. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660215. URL: http://doi.acm.org/10.1145/2660193.2660215.

[14] Asger Feldthaus et al. "Efficient Construction of Approximate Call Graphs for JavaScript IDE Services". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 752–761. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486887.

[15] José Fragoso Santos et al. "JaVerT: JavaScript Verification Toolchain". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 50:1–50:33. ISSN: 2475-1421. DOI: 10.1145/3158138. URL: http://doi.acm.org/10.1145/3158138.

[16] L. Haller et al. "Deciding floating-point logic with systematic abstraction". In: *2012 Formal Methods in Computer-Aided Design (FMCAD)*. Oct. 2012, pp. 131–140.

[17] Thomas A. Henzinger et al. "Abstractions from Proofs". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: ACM, 2004, pp. 232–244. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964021. URL: http://doi.acm.org/10.1145/964001.964021.

[18] Dongseok Jang and Kwang-Moo Choe. "Points-to Analysis for JavaScript". In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. Honolulu, Hawaii: ACM, 2009, pp. 1930–1937. ISBN: 978-1-60558-166-8. DOI: 10.1145/1529282.1529711. URL: http://doi.acm.org/10.1145/1529282.1529711.

[19] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type Analysis for JavaScript". In: *Static Analysis*. Ed. by Jens Palsberg and Zhendong Su. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 238–255. ISBN: 978-3-642-03237-0.

[20] Vineeth Kashyap et al. "JSAI: A Static Analysis Platform for JavaScript". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 121–132. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635904. URL: http://doi.acm.org/10.1145/2635868.2635904.

[21] Richard J. Orgass. "J. McCarthy. Towards a mathematical science of computation. Information processing 1962, Proceedings oflFIP Congress 62, organized by the International Federation for Information Processing, Munich, 27 August-1 September 1962, edited by Cicely M. Popplewell, North-Holland Publishing Company, Amsterdam1963, pp. 21–28. - John McCarthy. Problems in the theory of computation. Information processing 1965, Proceedings of IFIP Congress 65, organized by the International Federation for Information Processing, New York City, May 24–29, 1965, Volume I, edited by Wayne A. Kalenich, Spartan Books, Inc., Washington, D.C., and Macmillan and Co., Ltd., London, 1965, pp. 219–222." In: *Journal of Symbolic Logic* 36.2 (1971), pp. 346–347. DOI: 10.2307/2270319.

[22] Changhee Park and Sukyoung Ryu. “Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity”. In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 735–756. ISBN: 978-3-939897-86-6. DOI: 10.4230/LIPIcs.ECOOP.2015.735. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5245.

[23] Daejun Park, Andrei Stefănescu, and Grigore Roşu. “KJS: A Complete Formal Semantics of JavaScript”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. Portland, OR, USA: ACM, 2015, pp. 346–356. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737991. URL: http://doi.acm.org/10.1145/2737924.2737991.

[24] Aseem Rastogi et al. “Safe & Efficient Gradual Typing for TypeScript”. In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’15. Mumbai, India: ACM, 2015, pp. 167–180. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676971. URL: http://doi.acm.org/10.1145/2676726.2676971.

[25] Grigore Roșu and Traian Florin Șerbănuță. “An overview of the K semantic framework”. In: *The Journal of Logic and Algebraic Programming* 79.6 (2010). Membrane computing and programming, pp. 397–434. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2010.03.012. URL: http://www.sciencedirect.com/science/article/pii/S1567832610000160.

[26] Philipp Rümmer and Thomas Wahl. “An SMT-LIB Theory of Binary Floating-Point Arithmetic”. In: *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*. 2010.

[27] José Fragoso Santos et al. “Towards Logic-Based Verification of JavaScript Programs”. In: *Automated Deduction – CADE 26*. Ed. by Leonardo de Moura. Cham: Springer International Publishing, 2017, pp. 8–25. ISBN: 978-3-319-63046-5.

[28] José Santos et al. “JaVerT 2.0: Compositional Symbolic Execution for JavaScript”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 66:1–66:31. ISSN: 2475-1421. DOI: 10.1145/3290379. URL: http://doi.acm.org/10.1145/3290379.

[29] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. “Checking Safety Properties Using Induction and a SAT-Solver”. In: *Formal Methods in Computer-Aided Design*. Ed. by Warren A. Hunt and Steven D. Johnson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 127–144. ISBN: 978-3-540-40922-9.

[30] Manu Sridharan et al. “Correlation Tracking for Points-To Analysis of JavaScript”. In: *ECOOP 2012 – Object-Oriented Programming*. Ed. by James Noble. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 435–458. ISBN: 978-3-642-31057-7.

[31] Peter Thiemann. “Towards a Type System for Analyzing JavaScript Programs”. In: *Programming Languages and Systems*. Ed. by Mooly Sagiv. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 408–422. ISBN: 978-3-540-31987-0.

[32] Philipp Wendler. “Beiträge zu praktikabler Prädikatenanalyse”. In: *Ausgezeichnete Informatikdissertationen 2017*. Ed. by S. Hölldobler. Vol. D-18. LNI. Gesellschaft für Informatik (GI), 2018, pp. 261–270. ISBN: 978-3885799771. DOI: 20.500.12116/19476. URL: https://www.sosy-lab.org/research/phd/wendler/.