JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von

Christoph Kröll

IMPLEMENTATION OF THE ECMASCRIPT PROPOSAL NEW SET METHODS FOR GRAAL.JS

Bachelor Thesis with Oracle Labs

Angefertigt am

Institut für Systemsoftware

Betreuer

DI Sebastian Kloibhofer

Mitbetreuer

Dr. Christian Wirth
(Oracle Labs)

4 2021

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (BSc)

im Bachelorstudium

Informatik

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITÄT LINZ

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum

Ternberg, 21.4.2021

Unterschrift

Abstract

In recent years, the runtime environment GraalVM has been growing in popularity among the programming community, due to its improved performance compared to other Java Runtime Environments and its ability to handle more than one programming language at once. One of the languages that the GraalVM can process is JavaScript.

The implementation of JavaScript in the GraalVM follows the ECMAScript standard, which is constantly being improved upon via so called Proposals. These Proposals are contributed and developed by the community and checked by a committee until they're either discarded or accepted into the next version of ECMAScript.

This work is focused on enhancing the GraalVM by implementing one of the ECMAScript Proposal, New Set Methods. This Proposal is currently in Stage 2 and aims to add commonly used methods to the prototype of the Set datatype in order to decrease boilerplate code. In the first chapters, basic information about ECMAScript, the Set datatype, Graal.js and the Proposal New Set Methods is given. Following the introduction, this work explains how the Proposal was implemented and how the implementation was tested for correctness. At the end a benchmark is then given that compares the performance of the newly implemented methods in Graal.js with already existing libraries.

The result of the benchmark shows, that the implementation of the New Set Methods Proposal not only reduces the amount of boilerplate code, but that it will also improve the performance of JavaScript programs that use the new methods and run in Graal.js.

Kurzfassung

In den letzten Jahren hat die Laufzeitumgebung GraalVM unter Programmierenden, wegegn ihrer verbesserten Performance verglichen mit andenen Java Laufzeitumgebungen und ihrer Fähigkeit mehr als einer Programmiesprache zum selben Zeitpunkt umgehen zu können, an Popularität zugenommon. Eine der Sprachen, die die GraalVM ausführen kann, ist JavaScript.

Die Implementierung von JavaScript in der GraalVM folgt dem ECMAScript Standard, welcher fortgehenden über sogenannte Proposals verbesstert wird. Diese Proposals werden von der Community beigesteuert und entwickelt und von einem Komitee überprüft bis sie entweder verworfen oder in die nächste Version von ECMAScript aufgenommen werden.

Diese Arbeit ist auf die Verbesserung der GraalVM durch die Implementierung einer der ECMAScript Proposals, New Set Methods, fokusiert. Dieses Proposal ist derzeitig in Stage 2 und zielt darauf ab, allgemein gebräuchliche Methoden zum Prototypen des Set-Datentyps hinzuzufügen um den Boilerplatecode zu verringern. In den ersten Kapiteln werden grundlegende Informationen über ECMAScript, den Set-Datentyp, Graal.js und das Proposal New Set methods gegeben. Folgend der Enführung erklärt diese Arbeit wie das Proposal implementiert wirde und wie die Implementierung auf Korrektheit überprüft wurde. Am Ende wird eine Benchmark gelifert, die die Lesitung der neulich implementierten Methoden in Graal.hs mit bereits existierenden Bibliotheken vergleicht.

Das Resultat der Benchmark zeigt, dass die Implementierung des New Set Methods Proposals nicht nur den Boilerplate Code reduziert, sondern auch dass die Performance von JavaScript Propgrammen, die die neuen Methoden verwenden und in Graal.js laufen, verbessert wird.

List of Contents

1 Introduction

Java has long since been criticized for being slow, due to its interpreted performance. In time however, its speed has grown considerably thanks to countless optimizations, the most well-known of them being *Just in Time Compilation*. Yet the passage of time has not only been kind to Java. Many new features needed to be added to the programming language and the runtime environment grew cluttered.

Over the years, several research projects have been developed in order to improve the Java Virtual Machine. One such project was the Maxine Virtual Machine (MaxineVM). One goal of the MaxineVM was to investigate if it was possible to write an efficient Java Runtime Environment in Java.

Yet the MaxineVM was not the end of the development and a new Java Runtime Environment grew out of it, the Graal Virtual Machine, or GraalVM for short. The main argument that speaks for using the GraalVM is its performance compared to other Runtime Environments. Another argument is its ability to be able to process more than one programming language at a time. One such programming language is JavaScript.

JavaScript is a scripting language based on the ECMAScript specification and mainly used in web browsers and servers. ECMAScript is a standard developed by ECMA International via so called Proposals, which are constantly added to the specification. These Proposals introduce changes to the ECMAScript Standard and hence JavaScript such as adding new methods to extend its functionality, or to improve upon already existing parts by introducing new technologies.

In order to keep the GraalVM up to date, the goal of this thesis was to implement one such Proposal for Graal.js. Graal.js is the JavaScript implementation of the GraalVM. From the current Stage 2 Proposals [1], at least one Proposal needed to be chosen. Looking through the Proposals, one that did not require too much in-depth knowledge about JavaScript or the GraalVM and hence being a good entry point.

The result of this selection was the Proposal New Set Methods [2], a Proposal that intends to reduce the boilerplate code involving the `Set` datatype by adding new methods to the `Set` prototype.

2 Background

In this chapter, fundamental knowledge relevant to the thesis is presented. Section 2.1 ECMAScript explains what ECMAScript is, what relationship it has with JavaScript, and what Proposals are and their refinement. Section 2.2 Data Type Set gives information about what a Set is generally and in the specific case of JavaScript. Section 2.3 GraalVM introduces the reader to the Graal Virtual Machine, the Truffle Language Implementation Framework, a framework for enabling the execution of guest languages on the GraalVM, and Graal.js, the GraalVM's language implementation of JavasScript.

2.1 ECMAScript

ECMAScript is a standard developed for implementing an ECMA scripting language such as JavaScript. JavaScript is a programming language that is used mainly in web browsers and servers. *Since publication of the first edition in 1997, ECMAScript has grown to be one of the world's most widely used general-purpose programming languages. It is best known as the language embedded in web browsers but has also been widely adopted for server and embedded applications* [3].

2.1.1 Proposal Process

The ECMAScript Specification is developed and extended via Proposals. These Proposals are ideas specified and contributed by the community on how ECMAScript should be changed. Proposals undergo several stages during their development cycle, from Stage 0 to Stage 4. Once a Proposal has passed through Stage 4, it is accepted by the ECMAScript committee and will be put into the next version of the ECMAScript.

In Stage 0, Proposals have just been submitted and not yet been presented to the committee, or they have been shown to the committee, but the committee has not yet rejected the Proposal, or the Proposal still needs to meet some criteria to enter Stage 1.

Stage 1 Proposals are Proposal, that the committee has an interest in and intends to commit time to. In order to reach Stage 2, a specification needs to be written and added to the Proposal.

Stage 2 Proposals are Proposals, that the committee expects to be developed and eventually included in the standard. The major parts of the specification are finished, but issues, placeholders and TODO's might still be present.

Stage 3 Proposals are Proposals that are mostly finished but require actual implementation experience. In order to reach the final Stage, Stage 4, tests need to be written and actual implementations must exist.

A stage 4 Proposal is a Proposal that has met all qualifications and will be added to the next version of ECMAScript [4].

2.2 Data Type Set

The Set is one of the basic data structures used in Computer Science for organizing, managing, and storing data. In a set, every value is unique, hence if a value is added to a set, but is also contained within the set, the value will replace itself.

Sets are generally used for storing values that are unique, or for checking whether a given unique value is already contained within the set. Often, sets also implement performance optimizations, which increase the lookup speed. An example of such an improvement is the use of a hashing algorithm [5].

2.3 GraalVM

The GraalVM is a high-performance multilingual runtime intended to accelerate the execution of programs written in Java and other JVM languages. In addition, the GraalVM can also run applications written in other languages, such as JavaScript [6], Ruby [7], Python [8] and more. What is more, the GraalVM cannot only run programs in one single language, but it even allows the mixing of multiple programming languages in one application. This process is called polyglot programming or language interoperability [9].

2.3.1 Truffle API

The Truffle language implementation framework (henceforth "Truffle") is an open source library for building tools and programming languages implementations as interpreters for self-modifying Abstract Syntax Trees. Together with the open source GraalVM compiler, Truffle represents a significant step forward in programming language implementation technology in the current era of dynamic languages. [9]

The new implementation of the New Set Methods for the Set prototype heavily relies on the Truffle API. For instance, all the new methods added to the `Set.prototype` were implemented via nodes that inherit from the base node class

`com.oracle.truffle.api.nodes.Node` of the Truffle API. In addition, the code was further optimized using classes such as `BranchProfile` and `CompilerDirective`.

2.3.2 Graal.js

Graal.js [6] is the ECMAScript compliant, open source, runtime environment of the GraalVM for programs written in JavaScript. In addition to basic JavaScript, Graal.js also supports the Node.js runtime.

Previously, Graal.js already had an implementation of the `Set.prototype.` However, the methods described in the New Set Methods Proposal were still missing.

3 Proposal New Set Methods

The Set is a useful data structure in Javascript when unique values are required. A typical use case is to check if a given value is contained in a data structure. However, the functionality of the Set in JavaScript is limited. Basic operations from the Set Theory are missing, in particular, the *union*, *intersection*, *difference*, *symmetric difference*, *subset*, *superset* and *disjointed* operations [10].

These methods can be added to the prototype of the Set at runtime. There are even libraries that do exactly that, for instance Zet [11], Collections.js [12] and Immutable.js [13]. This can leads to boilerplate code. Hence is the motivation for this Proposal to reduce the amount of this boilerplate code, by implementing these operations natively [14].

3.1 Set.prototype.symmetricDifference(iterable)

The current specification of the New Set Method Proposal begins with the formal information as point 1 and can be ignored. The following eight points, 2-9, are the essential parts of the specification and describe the behavior the methods should implement in pseudocode.

The specification of one of these methods, `Set.prototype.symmetricDifference`, at point 5, is displayed in *Listing 1*. This method generates a `Set` that contains the values from both `Sets`, `set` and `newSet`, excluding all values that are found in `set` as well as `newSet`. Note, that the specification builds upon the present main specification of ECMAScript, such as `Type`, `Object`, `TypeError`, `SpeciesConstructor`, etc.

1. Let *set* be *this* value.
2. If *Type(set)* is not *Object*, throw a *TypeError* exception.
3. Let *Ctr* be ? *SpeciesConstructor(set %Set%)*.
4. Let *newSet* be ? *Construct(Ctr, set)*.
5. Let *remover* be ? *Get(newSet, "delete")*.
6. If *IsCallable(remover)* is *false*, throw a *TypeError* exception.
7. Let *adder* be ? *Get(newSet, "add")*.
8. If *IsCallable(adder)* is *false*, throw a *TypeError* exception.
9. Let *iteratorRecord* be ? *GetIterator(iterable)*.
10. Repeat,
    a. Let *next* be ? *IteratorStep(iteratorRecord)*.
    b. If *next* is false, return *newSet*.
    c. Let *nextValue* be ? *IteratorValue(next)*.
    d. Let *removed* be *Call(remover, newSet, « nextValue »)*.
    e. If *removed.[[Value]]* is *false*,
        i. Let *status* be *Call(adder, newSet, « nextValue »)*.
        ii. If *status* is an abrupt completion, return ? *IteratorClose(iteratorRecord, status)*.

Listing 1– Set.prototype.symmetricDifference(iterable) – Specification [14]

3.2 addEntryFromIterable(target, iterable, adder)

The method `addEntryFromIterable` is the last method defined in the New Set Methods Proposal and is used only as a helper method. The method intends to add all the values from an `iterable` using the given `adder` function to the `target`. The `addEntryFromIterable` method is used in two other methods, `union` and `isSubsetOf`. The specification is as follows:

1. If *isCallable(adder)* is false, throw a *TypeError* exception.
2. Let *iteratorRecord* be ? *GetIterator(iterable)*.
3. Repeat,
    a. Let *next* be ? *IteratorStep(iteratorRecord)*.
    b. If *next* is false, return.
    c. Let *nextValue* be ? *IteratorValue(next)*.
    d. Let *status* be *Call(adder, target, « nextValue »)*.
    e. If *status* is an abrupt completion, return ? *IteratorClose(iteratorRecord, status)*.

Listing 2– addEntryFromIterable(target, iterable, adder) Specification [2]

3.3 Difficulties

During execution, of the unit tests for the method `symmetricDifference`, a problem occurred in one of the edge cases, specifically when testing if a `TypeError` is thrown if the add function of the `Set` prototype is not callable. A look at line 6 of the execution of the `Set` constructor in the ECMAScript specification shown in Listing 3revealed the problem.

```
1.  If NewTarget is undefined, throw a TypeError exception.
2.  Let set be ? OrdinaryCreateFromConstructor(NewTarget,
   "%Set.prototype%", « [[SetData]] »)
3.  Set set.[[SetData]] to a new empty List.
4. If iterable is either undefined or null, return set.
5. Let adder be ? Get(set, "add").
6. If IsCallable(adder) is false, throw a TypeError exception.
7. Let iteratorRecord be ? GetIterator(iterable).
8. Repeat,
      a. Let next be ? IteratorStep(iteratorRecord).
      b. If next is false, return set.
      c. Let nextValue be ? IteratorValue(next).
      d. Let status be Call(adder, set, « nextValue »).
      e. If status is an abrupt completion, return
         ? IteratorClose(iteratorRecord, status).
```

Listing 3– Set.prototype.constructor Specification [3]

During the execution of the constructor, the program already checks if the add function of the `Set` is callable. If that is not the case, a `TypeError` is thrown. This means, that the `isCallable` check in the method `symmetricDifference` for the add function is redundant, because the exception would already have been thrown in the constructor.

However, to be on the safe side, the check is kept in the `symmetricDifference` method and a comment was added, telling readers that it cannot be reached due to the constructor call.

4 Implementation

This chapter focuses on the actual implementation of the Proposal in Graal.js. Section *4.1 Parser* informs the reader why no changes to the parser were necessary, while Section *4.2 Nodes* tells the reader about the nodes that were added and/or changed. Finally, Section *4.3 Optimization* goes over how the code was further optimized.

Since the focus of the implementation lay in adding new methods to the prototype of the Set Datatype, there were no new additions to the parser. The only Java file that needed to be changed and extended was the `SetPrototypeBuiltins.java` file.

To implement the methods specified in the New Set Methods Proposal, new nodes for each of the given methods needed to be written in the `SetPrototypeBuiltins` class. The first of the new nodes was the `JSSetNewSetMethod` node, which acts as the parent class for all the other nodes and offers a variety of fields and methods that the other nodes require. Then the `union`, `intersection`, `difference`, `symmetricDifference`, `isSubsetOf`, `isSupersetOf` and `isDisjointedFrom` nodes followed. Each of these nodes implements the behaviour of the associated method from the New Set Methods Proposal.

In order to make the new methods accessible, new entries were added to the `SetPrototype` enum. Then, using the new entries, the `createNode` method of the `SetPrototypeBuiltins` was extended, so that the nodes for the new set methods were available.

4.1 Parser

The `Set` data type was introduced in the 6th Edition of ECMAScript in 2015 and has since then been a part of Javascript [15]. Hence the `Set` has already been supported by Graal.js and no new prototype needed to be added. Only the current prototype of the `Set` needed to be changed and extended. In addition Graal.js already had all the necessary code for parsing the newly added methods, so no alterations to the parser were required.

4.2 Nodes

Eight new nodes have been added to the `SetPrototypeBuiltins` class. The `JSSetNewSetMethod` is the super class of all given function nodes and implements the

common functionality, such as the helper method `addEntryFromIterable` and several nodes or objects to optimize access to certain data.

Then there are the seven method nodes `JSSetUnionNode`, `JSSetInterscetionNode`, `JSSetDifferenceNode`, `JSSetSymmetricDifferenceNode`, `JSSetIsSubsetOfNode`, `JSSetIsSupersetOfNode`, and `JSSetIsDisjointedFromNode`, each implementing one of the new methods of the `Set.prototype`.

4.2.1 JSSetNewSetMethod

Though there already was the `JSSetOperation` node, a new superclass, `JSSetNewOperation`, was introduced for the methods introduced with the New Set Method Proposal. There were two main reasons for making a new class, instead of adding new code to the `JSSetOperation`.

First, all the new methods that need to be coded require several helper nodes used for iteration. Other `Set` methods such as `add`, `clear`, or `delete` do not need them. Hence the overhead for these methods is reduced.

Second, the Proposal New Set Methods is still in Stage 2 and changes to the specification might be possible. Hence, if there are alterations, having everything separate from the base, might make it easier to realize these changes.

Beyond this, the `JSSetNewOperation` has several tasks to fulfill:

- Implementation of the helper method `addEntryFromIterable(target, iterable, adder)` as specified in the New Set Methods Proposal.
- Access to the nodes `GetIteratorNode`, `IteratorStepNode`, `IteratorValueNode`, and `IteratorCloseNode`, as well as the helper method `iteratorCloseAbrupt`.
- Access to common `BranchProfiles` which are used in the `JSSetNewOperation` node but also in subclasses.
- Provide optimized getters for the `add`, `delete` and `has` method of the `Set`.
- Provide optimized helper methods `call` and `isCallable`.
- Provide a method `constructSet` for constructing new `Sets`.

Helper Method – addEntryFromIterable

First, the `JSSetNewOperation` node implements the method from the New Set Methods Proposal. This method is later used by the in sections *4.2.2 JSSetUnionNode* and *4.2.6 JSSetIsSubsetOfNode* in order to add all the elements from an iterable value to a given target.

Iteration Support

The second purpose of the `JSSetNewOperation` is to allow access to the iteration nodes. The `GetIteratorNode` is used for getting the `IteratorRecord` from an iterable value. The `IteratorStepNode` is used for checking whether the end of the iteration has been reached. The `IteratorValueNode` is used for getting the next value in the iterator and the `IteratorCloseNode` is used to close the iterator in the case that an error has occurred. There is also the `iteratorCloseAbrupt` method, a method that is called whenever an iterator needs to be closed abruptly, such as when an error occurs during an iteration. This is something that must be checked for in all the given methods from the New Set Methods Proposal.

Branch Profiles

`BranchProfiles` are objects from the Truffle Framework, which are used for optimizing the code by removing a piece of code until a given branch has been entered. `iteratorError` and `adderError` are also accessible to the inheriting nodes. `iteratorError` is a `BranchProfile` that should be entered whenever an error occurs during iteration and `adderError` should be entered whenever an error occurs with the add method, such as the add method not being callable. Both the `iteratorError` and `adderError` `BranchProfiles` are used in the helper method `addEntryFromIterable` as well as the sub nodes.

Optimized Getters

The get methods, `getAddFunction(target)`, `getRemoveFunction(target)`, and `getHasFunction(target)` use the `PropertyGetNodes` `getAddNode`, `getRemoveNode`, and `getHasNode` respectively in order to get the add, remove, and has functions from a given object. Each of these nodes is instantiated via *Lazy Initialization* and removes the code of the *Lazy Initialization* from the code once it has been

instantiated via the call `CompilerDirectives.transferToInterpreterAndInvalidate()`.

Optimized Function Calls

The `call(function, target, userArguments)` method is used for executing a given callable, `function`, with the given `target` object as the `this` value and the given arguments. This method uses the `JSFunctionCallNode`, `callFunctionNode`, which is initialized using the same process of *Lazy Initialization* as the get methods.

Constructor Support

The `JSSetNewOperation` node contains the method `constructSet(Object…)` used for constructing a new set, with or without input parameters. The method `constructSet` is used in the `union`, `intersect`, `difference` and `symmetricDifference` methods.

4.2.2 JSSetUnionNode

The `JSSetUnionNode` implements the behavior of the `Set.prototype.union(iterable)` method. The result should be a new `Set` that contains all the values from the current `Set` (`this` value) and the `iterable`. See *Figure 1* for a depiction of the union method. All values from both input sets are contained in the output set. *Listing 4* gives a more specific example of how the union method would be used in Pseudocode.

```
let set1 = new Set(0, 1, 2)
let set2 = new Set(2, 3, 4)
let union = set1.union(set2) // {0, 1, 2, 3, 4} expected
```

Listing 4 – union pseudocode Example

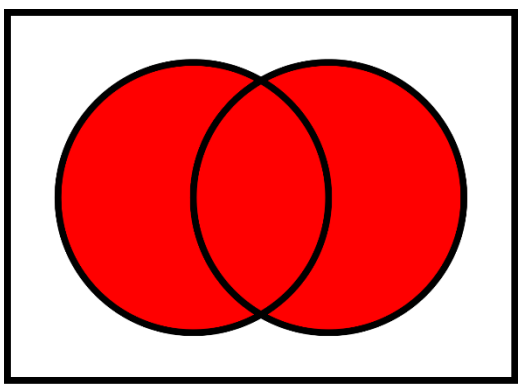

Figure 1 – union Venn Diagramm

When the `union` method is called, it first checks if the given current object `(this value)` is a `Set`, via a given guard using the method `isJSSet(set)`. If the current object is not a `Set`, then a `TypeError` will be thrown.

Next, a new set, called `newSet`, is created by using the `constructSet(set)` method. This new set contains all the values from the current `set (this value)`. After the new set has been created, the add function is retrieved from it, using the `getAddFunction` and stored in a variable called adder.

Using the `newSet`, `iterable` and the `adder`, the method `addEntryFromIterable(target, iterable, adder)` is now called in order to add all the values from the `iterable` to the `newSet`. Hence, the `newSet` now contains both the values from the current `set` and the `iterable` and is returned.

4.2.3 JSSetIntersectionNode

The `JSSetIntersectionNode` implements the behavior of the `Set.prototype.intersection(iterable)` method. The result of this method should be a new `Set` containing all values that were both in the current `Set` (`this value`) and the `iterable`. *Figure 2* demonstrates how the output set only contains the values of both input sets while *Listing 5* gives a concrete example in Pseudocode.

```
let set1 = new Set(0, 1, 2)
let set2 = new Set(2, 3, 4)
let intersection = set1.intersect(set2) // {2} expected
```

Listing 5 – intersect pseudocode Example

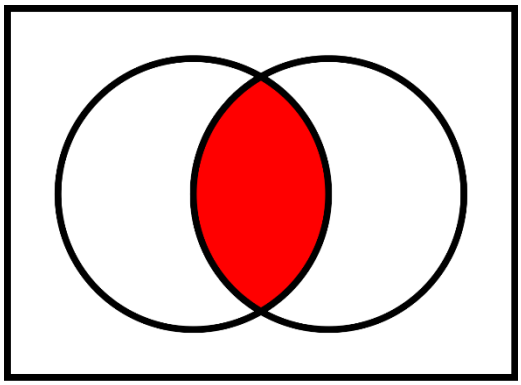

Figure 2 – intersect Venn Diagramm

When the `intersect` method is called, it first checks if the given current object `(this value)` is a `Set`, via a given guard using the method `isJSSet(set)`. If the current object is not a `Set`, then a `TypeError` will be thrown.

Next, a new empty set, called `newSet`, is created by using the `constructSet(set)` method. Then, the `has` function from the current set (`this`) and the `add` function from the `newSet` are retrieved using the methods `getHasFunction(set)` and the `getAddFunction(newSet)`. If one of the retrieved functions is not callable, then a `TypeError` will be thrown.

Next, the `iteratorRecord` of the `iterable` is fetched using the `GetIteratorNode`. With the `iteratorRecord`, the `iterable` is iterated over in an endless while loop, surrounded by a try catch, that can be triggered by any exception. If an exception occurs, the iterator will be closed abruptly using the `iteratorCloseAbrupt` method and the exception thrown.

Inside the endless while loop, the program first checks if the iterator has a next value using the `iteratorStepNode`. If the result is false and the iterator has no next value, then the `newSet` is returned. If the result is true, then the `nextValue` of the iterator is retrieved using the `iteratorValueNode`.

The method then checks if the retrieved `nextValue` is contained within the current `Set` (`this value`). If the `nextValue` is in the current `Set`, it is added to the `newSet`.

4.2.4 JSSetDifferenceNode

The `JSSetDifferenceNode` implements the behavior of the `Set.prototype.difference(iterable)` method. The result of this method should be a new `Set` containing all values that were in the current `Set` (`this value`), but not in the `iterable`. See *Figure 3* of how the output set contains only the values that were in one set but not in the other. For a more specific example see *Listing 6*.

```
let set1 = new Set(0, 1, 2)
let set2 = new Set(2, 3, 4)
let difference = set1.difference(set2) // {0, 1} expected
```

Listing 6 – difference pseudocode Example

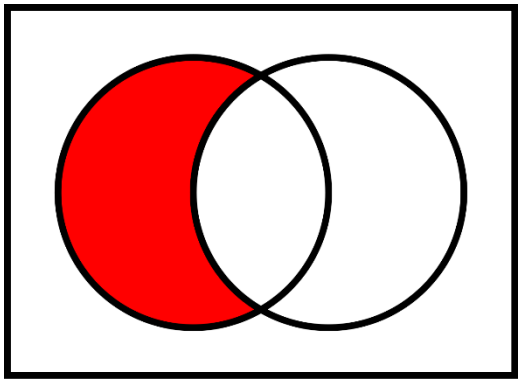

Figure 3 – difference Venn Diagramm

When the `difference` method is called, it first checks if the given current object `(this value)` is a `Set`, via a given guard using the method `isJSSet(set)`. If the current object is not a `Set`, then a `TypeError` will be thrown.

A new set called `newSet` is created by using the `constructSet(set)` method. This new set contains all the values from the current `set (this value)`. Once the new set has been created its remove function is retrieved using the `getRemoveFunction` and then stored in the variable `remover`. If the remove function is not callable, a `TypeError` is thrown.

Next the `iteratorRecord` of the `iterable` is retrieved using the `GetIteratorNode`. Using this iterator, the `iterable` is iterated over in an endless while loop, surrounded by a try catch, that can be triggered by any exception. If an exception occurs, the iterator will be closed abruptly using the `iteratorCloseAbrupt` method and the exception thrown.

In the endless while loop, the program first checks if the iterator has a next value using the `iteratorStepNode`. If the result is false and the iterator has no next value, then the `newSet` is returned. If the result is true, then the `nextValue` of the iterator is retrieved using the `iteratorValueNode`.

The method then removes the retrieved `nextValue` from the `newSet` using the `remover`.

4.2.5 JSSetSymmetricDifferenceNode

The `JSSetSymmetricDifferenceNode` implements the behavior of the `Set.prototype.difference(iterable)` method. The result of this method should be a new `Set` containing all the values that were in the current `Set` (`this value`) and the `iterable`, but not in both. *Figure 4* displays the output of the symmetric difference, containing all the values of the union of the two input sets without the values of the intersection. For a more in-depth example, see Listing 7.

```
let set1 = new Set(0, 1, 2)
let set2 = new Set(2, 3, 4)
let symDif = set1.symmetricDifference(set2) // {0, 1, 3, 4} expected
```

Listing 7 – symmetricDifference pseudocode Example

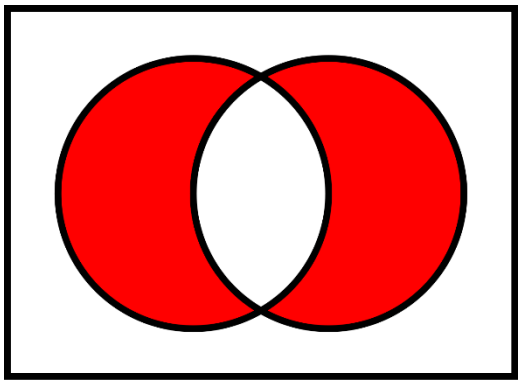

Figure 4 - symmetricDifference Venn Diagramm

When the `symmetricDifference` method is called, it first checks if the current object `(this value)` is a `Set`, via a given guard using the method `isJSSet(set)`. If the current object is not a `Set`, then a `TypeError` will be thrown.

Next, a new set, called `newSet`, is created by using the `constructSet(set)` method. This new set contains all the values from the current `set (this value)`. Then the remove and add function are retrieved from the `newSet`, using the `getRemoveFunction` and `getAddFunction`. If either of the two functions is not callable, a `TypeError` is thrown.

Subsequently the `iteratorRecord` of the `iterable` is retrieved using the `GetIteratorNode`. Using this iterator, the `iterable` is iterated over in an endless while loop, surrounded by a try catch, that can be triggered by any exception. If an exception occurs, the iterator will be closed abruptly using the `iteratorCloseAbrupt` method and the exception thrown.

During each iteration of the endless while loop, the program first checks if the iterator has a next value using the `iteratorStepNode`. If the result is false and the iterator has no next value, then the `newSet` is returned. If the result is true, then the `nextValue` of the iterator is retrieved using the `iteratorValueNode`.

Following this step, the `nextValue` is removed from the `newSet` and the return value of the remove function is stored in the variable `removed`. If the value of `removed` is false, the `nextValue` is added to the `newSet`.

4.2.6 JSSetIsSubsetOfNode

The `JSSetIsSubsetOfNode` implements the behavior of the `Set.prototype.isSubsetOf(iterable)` method. The result of this method is a boolean value that is true if all elements of the current `Set` (`this`) are contained within the `iterable`. *Listing 8* demonstrates how the `isSubsetOf` method might be used and its results.

```
let set1 = new Set(0, 1, 2)
let set2 = new Set(0, 1, 2, 3, 4, 5)
let isSubset = set1.isSubsetOf(set2) // true expected
```

Listing 8 – isSubsetOf pseudocode Example

First the `isSubsetOf` method checks if the current object `(this value)` is a `Set`, via a given guard using the method `isJSSet(set)`. If the current object is not a `Set`, a `TypeError` is thrown.

Next, the `iteratorRecord` of the `set` is retrieved using the `GetIteratorNode`.

Following this, the method checks if the given `iterable` is an object. If not a `TypeError` is thrown. Once the `iterable` has been determined to be an object, a variable called `otherSet` is assigned the value of the `iterable`.

Next the has function of the `otherSet` is retrieved. If the `has` function is not callable, a newly `constructed Set` is assigned to the variable `otherSet`. The method `addIterable(otherSet, iterable, getAddFunction(otherSet))` is called in order to add the values from the `iterable` to the `otherSet` and then the `has` function of the `otherSet` is retrieved.

With the `iteratorRecord`, the `set` is iterated over in an endless while loop, surrounded by a try catch, that can be triggered by any exception. If an exception occurs, the iterator will be closed abruptly using the `iteratorCloseAbrupt` method and the exception thrown.

Inside the endless while loop, the program first checks if the iterator has a next value using the `iteratorStepNode`. If the result is false and the iterator has no next value, then true is returned. If the result is true, then the `nextValue` of the iterator is retrieved using the `iteratorValueNode`.

Finally, the program checks if the `nextValue` is contained in the current `Set` (`this value`). If not, false is returned.

4.2.7 JSSetIsSupersetOfNode

The `JSSetIsSupersetOfNode` implements the behavior of the `Set.prototype.isSupersetOf(iterable)` method. The result of this method is a boolean value that is true if all elements of the `iterable` are contained in the current `Set` (`this value`). A concrete example of how this method would be used and its output is shown in *Listing 9*.

```
let set1 = new Set(0, 1, 2, 3, 4, 5)
let set2 = new Set(0, 1, 2)
let isSuperset = set1.isSupersetOf(set2) // true expected
```

Listing 9 – isSupersetOf pseudocode Example

At the beginning the `isSupersetOf` method checks if the current object `(this value)` is a `Set`, via a given guard using the method `isJSSet(set)`. If the current object is not a `Set`, a `TypeError` is thrown.

Once it has been determined that the current object is a `Set`, the `has` function is retrieved from it. If the retrieved value is not callable, a `TypeError` is thrown.

Then the `iteratorRecord` of the `iterable` is retrieved using the `GetIteratorNode`. Using this iterator, the `iterable` is iterated over in an endless while loop, surrounded by a try catch, that can be triggered by any exception. If an exception occurs, the iterator will be closed abruptly using the `iteratorCloseAbrupt` method and the exception thrown.

In the endless while loop, the program first checks if the iterator has a next value using the `iteratorStepNode`. If the result is false and the iterator has no next value, then true is returned. If the result is true, then the `nextValue` of the iterator is retrieved using the `iteratorValueNode`.

During the last part of the loop, the program checks if the `nextValue` is contained in the current `Set` (`this value`). If not, false is returned.

4.2.8 JSSetIsDisjointedFromNode

The `JSSetIsSupersetOfNode` implements the behavior of the `Set.prototype.isDisjointedFrom(iterable)` method. The result of this method is a boolean value that is true if no element of the current `Set` (`this value`) is contained in the `iterable` and vice versa. *Listing 10* presents a specific example of how this method might be used.

```
let set1 = new Set(0, 1, 2)
let set2 = new Set(3, 4, 5)
let isDisjointed= set1.isDisjointedFrom(set2) // true expected
```

Listing 10 – isDisjointedFrom pseudocode Example

At the beginning, the `isDisjointedFrom` method checks if the current object `(this value)` is a `Set`, via a given guard using the method `isJSSet(set)`. If the current object is not a `Set`, a `TypeError` is thrown.

Once it has been determined that the current object is a `Set`, its `has` function is retrieved. If the retrieved `has` function is not callable, a `TypeError` is thrown.

Next the `iteratorRecord` of the `iterable` is retrieved using the `GetIteratorNode`. Using this iterator, the `iterable` is iterated over in an endless while loop, surrounded by a try catch, that can be triggered by any exception. If an exception occurs, the iterator will be closed abruptly using the `iteratorCloseAbrupt` method and the exception thrown.

In the endless while loop, the program first checks if the iterator has a next value using the `iteratorStepNode`. If the result is false and the iterator has no next value, then true is returned. If the result is true, then the `nextValue` of the iterator is retrieved using the `iteratorValueNode`.

Finally, the program checks if the `nextValue` is contained in the current `Set` (`this value`). If so, false is returned.

4.3 Optimization

From the moment the methods for the New Set Methods Proposal were finished, changes have been made. Parts were revised and added in order to improve both the code's readability and performance.

There were two main ways that this was done. First, for all branches that throw exceptions `BranchProfiles` were created and added. Second, methods from JSRuntime and JSObject were replaced with their node counterparts. While the first variant introduces a runtime call to the code, having nodes allows the GraalVM to optimize the implementation by caching on data types and inputs.

4.3.1 BranchProfile

BranchProfiles are profiles to speculate on branches that are unlikely to be visited. If the BranchProfile.enter() method is invoked first the optimized code is invalidated and the branch where BranchProfile.enter() is invoked is enabled for compilation. Otherwise, if the BranchProfile.enter() method was never invoked the branch will not get compiled [16].

In simple terms, this means that `BranchProfiles` should be used at passages of code that have a low chance of being executed. This addition to the code should reduce the compile time and the amount of generated machine code, if used correctly.

An example for the usage of `BranchProfiles` would be when throwing exceptions. Putting both the throw statement and the Error object creation, neither of them being cheap, behind a `BranchProfile`, removes a significant amount of the generated machine code. In the case of the New Set Methods Proposal, the `BranchProfile` would for instance be used when `TypeError` exceptions are thrown, since the chance of the error occurring in production code should be low.

4.3.2 Nodes

In order to increase the performance of the methods from the Proposal, runtime methods, which are methods that have no knowledge about the given object and hence cannot be optimized, were replaced by nodes.

For instance, the method `JSObject.get(obj, propertyName)` was replaced by `PropertyGetNodes`. This was done, because the `JSObject.get` method must look up the sought-after property in the memory every time, whereas a `PropertyGetNode` uses Inline Caching in order to improve the search performance. Inline Caching is a technique used for increasing the performance of said lookups, by remembering previous results.

The same goes for the methods `JSRuntime.call` and `JSRuntime.isCallable`, which have been replaced by the `JSFunctionCallNode` and `IsCallableNode`.

5 Unit Tests

This section tells the reader about the unit tests used to test the newly implemented functionality for correctness. Section *5.1 Available Tests*, talks about why no existing tests could be used to explicitly check for the correctness of the new methods, while section *5.2 Tests Contributed by this Work* presents the unit tests written for validating the implementation of the Proposal to the reader.

In order to determine the correctness of the newly implemented functionality, new tests needed to written. These tests check for the correct output during a normal runtime, where no errors should occur, as well as for all edge cases and errors that may be thrown. Graal.js then uses all its tests before every merge of new code in order to ensure that all changes to the project are correct and that the functionality has not regressed, i.e. changes do not conflict with the already existing functionality.

5.1 Available Tests

Since the implementation of the New Set Methods Proposal required that only new methods be implemented and added to the runtime, currently avaiblable tests could not be used for testing the new methods. Neither did the New Set Methods Proposal provide any tests, which it requires to move up to Stage 4. However, previously existing tests were used as reference for writing the new tests. In addition, the already existing tests helped to verify that no functionality has regressed.

5.2 Tests Contributed by this Work

Unit tests were written for all of the newly implemented methods, `union`, `intersect`, `difference`, `symmetricDifference`, `isSubsetOf`, `isSupersetOf` and `isDisjointedFrom`. For each of these methods several unit tests have been written. These tests verify the result of a given method, check for edge cases and exceptions that may be thrown.

5.1 Helper Methods

Before looking at the actual unit tests, it is important that the following helper methods be mentioned first. These helper methods are used as aids for constructing the JavaScript code that will later be evaluated in the actual unit tests.

```
private static String createSetString(int... values) {
  return "new Set(" + Arrays.toString(values) + ")";
}
```

Listing 11 – createString helper method

The helper method `createSetString` receives an array of integer values and returns a string representing a piece of code that generates a new `Set`. The newly generated set should contain all the integer values that were passed as parameters.

```
private static final String setEqualsString(String varName1,
    String varName2) {
  return String.format("%s.size === %s.size && [...%s].every(value =>" +
          "%s.has(value));", varName1, varName2, varName1, varName2);
}
```

Listing 12 – setEqualsString helper method

The helper method `setEqualsString` receives the names of two variables, `varName1` and `varName2`, and should return a piece of code that checks if the two given `Sets` behind the variable names `varName1` and `varName2` are equal.

In order to test the equality of the two `Sets`, the number of elements of the two given `Sets` is compared first. If the two `Sets` do not have the same number of elements, they are not equal. If the two `Sets` have the same number of elements, the elements are compared next. If every value of the one `Set` is contained in the second `Set`, they are equal. No further checks are required because no duplicates can exist in a Set.

5.2 Symmetric Difference

Since a problem appeared while testing the `Set.prototype.symmetricDifference` method, some of its test methods will now be shown.

```
@Test
public void testSymmetricDifference() {
  try (Context context = JSTest.newContextBuilder().option(
      JSContextOptions.ECMASCRIPT_VERSION_NAME, "2022").build()) {
    String code = String.format("var set1 = %s; var set2 = %s;" +
            "var expected = %s;" +
            "var result = set1.symmetricDifference(set2); %s;",
            createSetString(1, 2, 3, 4), createSetString(3, 4, 5, 6),
            createSetString(1, 2, 5, 6),
            setEqualsString("result", "expected"));
    Value result = context.eval(JavaScriptLanguage.ID, code);
    assertTrue(result.isBoolean());
    assertTrue(result.asBoolean());
  }
}
```

Listing 13 – testSymmetricDifference Unit Test

The unit test, `testSymmetricDifference`, is a test for the default case. It starts by creating a new `context` that a given piece of JavaScript code will be executed in. Since the New Set Methods Proposal is still at Stage 2 and it is unknown when it will be finished, it is only available for ECMAScript version 2022, which currently stands for a yet unpublished version of ECMAScript.

After the `context` has been generated, the `code` that will be executed is put together next using `String.format` and two helper methods `createSetString` and `setEqualsString`. The given `code` should create two new Sets, `set1` with the values {1, 2, 3, 4,} and `set2` with the values {3, 4, 5, 6}. Then a new `Set`, `expected`, with the values {1, 2, 5, 6} is created. This `Set` represents the expected result of the `symmetricDifference` method between `set1` and `set2`. Next, a `Set result` is created, which contains the actual values of the intsersection of `set1` and `set2`. Finally, the expected `Set` and the resulting `Set` are compared with each other and the result of the comparison is returned.

If the returned `result` of the executed code is a boolean value and if the given boolean value is `true`, then the test passes.

```
@Test
public void testSymmetricDifferenceAddNotCallable() {
  try (Context context = JSTest.newContextBuilder().option(
      JSContextOptions.ECMASCRIPT_VERSION_NAME, "2022").build()) {
    String code = String.format("var set1 = %s; var set2 = %s;" +
            "Set.prototype.add = 666;" +
            "set1.symmetricDifference(set2);",
            createSetString(1, 2, 3, 4), createSetString(3, 4, 5, 6));
    context.eval(JavaScriptLanguage.ID, code);
    Assert.fail("Non callable expected.");
  } catch (PolyglotException ex) {
    assertTrue(ex.getMessage().contains("TypeError:"));
  }
  try (Context context = JSTest.newContextBuilder().option(
      JSContextOptions.ECMASCRIPT_VERSION_NAME, "2022").build()) {
    String code = String.format("var set1 = %s; var set2 = %s;" +
            "delete(Set.prototype.add);" +
            "set1.symmetricDifference(set2);",
            createSetString(1, 2, 3, 4), createSetString(3, 4, 5, 6));
    context.eval(JavaScriptLanguage.ID, code);
    Assert.fail("Non callable expected.");
  } catch (PolyglotException ex) {
    assertTrue(ex.getMessage().contains("TypeError:"));
  }
}
```

Listing 14 – testSymmetricDifferenceAddNotCallable Unit Test

The unit test from Listing 14 should have checked if a `TypeError` is thrown in the case that the `add` method of the `Set` is a non-callable value by replacing the add function with the integer value 666 or deleting the add function. `TypeErrors` are indeed thrown in both cases once the code is executed. However, the `TypeError` is not thrown from the intended branch, due to an inconsistency in the specification of the `Set` constructor and the `symmetricDifference` method.

6 Benchmark

In this chapter, the implementation of the New Set Methods Proposal is tested for performance. The results from before the optimization are then compared with the results after, and the results of another, third party library written in JavaScript that was then executed in Node.js. Section *6.1 Implementation* gives a short explanation how the Benchmark works, while section *6.2 Results* presents the results from before and after the optimization to the reader and section *6.3 Comparison* shows the results of the benchmark in Node.js [17] using a third party library.

Because Sets are an important data type and the newly implemented methods may be used in performance critical applications, a benchmark was written in order to measure how the performance improves or worsens depending on changes to the code.

6.1 Implementation

The benchmark is written in JavaScript. A given method is used several times, and the time it took to execute the given method is added to an array. In the end, the total time, average time, and median are calculated using the array and printed on the console.

6.2 Results

The results of the benchmark are obviously in favor of the optimizations using the BranchProfile and Nodes.

METHOD	TOTAL TIME(MS)	AVERAGE TIME(MS)	MEDIAN(MS)
UNION	29187	291.87	194
INTERSECT	18805	188.05	170
DIFFERENCE	19812	198.12	178
SYMMETRICDIFFEREMCE	21842	218.42	199
ISSUBSETOF	10646	106.46	92
ISSUPERSETOF	9930	99.3	92
ISDISJOINTEDFROM	5421	54.21	52

Table 1 – Benchmark Before Optimization

METHOD	TOTAL TIME(MS)	AVERAGE TIME(MS)	MEDIAN(MS)
UNION	40062	400.62	150
INTERSECT	12837	128.37	114
DIFFERENCE	15607	156.07	136
SYMMETRICDIFFEREMCE	16068	160.68	147
ISSUBSETOF	8245	82.45	70
ISSUPERSETOF	7507	75.07	70
ISDISJOINTEDFROM	5079	50.79	48

Table 2 – Benchmark After Optimization

When comparing *Table 1* and *Table 2*, the result that stands out the most, is the difference of the required time in the union function. Before the optimization, the total time it took to execute the test was about 30 seconds. After the optimization, the test took close to 11 seconds longer. This also translates to the average time, where the performance visibly decreased by 100 milliseconds.

However, comparing the median values reveals, that the performance might have improved by about a fourth. Taking a more in depth look at the actual test results for each iteration the reason becomes obvious. The first few executions of the union method took about one up to five seconds. However, after the first 50 executions, the execution time has dropped down to about 150 milliseconds.

For all other methods however, the performance increased by about 20% -30%.

6.3 Comparison

To get a better grasp of the effect the optimizations of the GraalVM and Graal.js have on the performance of the `Set` methods, they were tested in another runtime environment. However, because neither *V8* [18] nor *SpiderMonkey* [19] have implemented the methods right now, the only remaining option left was to run the benchmark in the original *Node.js* [17] using the polyfill *Javascript Set Extensions* [20].

Unfortunately, one of the methods, `isDisjointedFrom`, was missing from the polyfill, hence it could not be measured. The intersect method, that was called `intersection` in the New Set Methods Proposal, is called `intersect` in the *Javascript Set Extensions*. Moreover, two of the methods, `difference` and `symmetricDifference`, were not

part of the `Set.prototype`, but rather the Set instead. Hence, some parts of the benchmark needed to be adapted, although the changes should only have a minor impact on the performance.

METHOD	TOTAL TIME(MS)	AVERAGE TIME(MS)	MEDIAN(MS)
UNION	26923	269.23	265
INTERSECT	24822	248.22	239
DIFFERENCE	18787	187.87	186
SYMMETRICDIFFEREMCE	28040	280.40	274
ISSUBSETOF	13087	130.87	129
ISSUPERSETOF	13243	132.43	132

Table 3 - Node.js Benchmark using "js-set-extensions"

As can be seen in *Table 3*, the polyfill performed significantly worse than Graal.js. For instance, the execution of `symmetricDifference` method took about 1.75 times as long as in Graal.js. When comparing the medians of the `isSupersetOf` method the version of Graal.js is nearly two times as quick.

Such being the case, the implementation of the New Set Methods Proposal in Graal.js not only reduced the boilerplate code, as was the motivation of the Proposal, but it also increased the performance considerably.

7 Conclusion

A basic implementation of the Proposal New Set Methods in Graal.js required a basic familiarity with Java and JavaScript, as well as the skills to read and understand the ECMAScript specification. Through it one will learn about Graal.js' node system such as how one can add new methods or iterate over a value.

To verify the correctness of the implementation, it needed to be checked using unit tests. Unfortunately, neither Graal.js nor the Proposal provided any tests. As such, new tests needed to be thought of, that check for both the correctness of the program and every important edge case. During this process, it was discovered, that there was a conflict in the ECMAScript specification between the `Set.prototype.symmetricDifference` method and the `Set.prototype.constructor`, that lead to an unreachable statement in the code. The solution to solving this problem was to temporarily ignore it, since the correct exception was already thrown in the constructor and because the Proposal was still in Stage 2 and might be changed in the future.

After the code was checked successfully for its correctness, a benchmark was written to test the performance. This benchmark was then used on the current, non-optimized versions of the methods.

Once this was done, parts of the code needed to be improved upon. The first part of the optimization process was to add BranchProfiles for each type of error that could occur, such as the add function not being callable or an error occurring during the iteration. This addition reduced the amount of code that needed to be compiled at the start of the execution. Secondly, runtime calls were replaced by their node variants, allowing the GraalVM to further optimize the methods.

With the optimization being finished, the benchmark was executed once more, and the results were compared with the ones before. The results were in favor of the newer version, although the startup time of the methods might have increased due to the optimizations.

Finally, the results of the optimized benchmark were compared with the results of another runtime environment, specifically *Node.js*, using an already existing library, *Javascript-Set-Extensions*. The results of the benchmark strongly supported the implementation in Graal.js and the GraalVM.

8 List of References

[1] ECMA International, "ECMA Script Proposals," [Online]. Available: https://github.com/tc39/proposals. [Accessed 3 4 2021].

[2] ECMA International, "New Set Methods - Proposal," [Online]. Available: https://github.com/tc39/proposal-set-methods. [Accessed 5 4 2021].

[3] ECMA International, "ECMA Script Language Specification," 31 3 2021. [Online]. Available: https://262.ecma-international.org/.

[4] ECMA International, "The TC39 Process," [Online]. Available: https://tc39.es/process-document/. [Accessed 21 4 2021].

[5] D. R. Sheehy, "A First Course on Data Structures in Python," 2019, pp. 19-20.

[6] Oracle, Labs, "Graal.js," [Online]. Available: https://github.com/oracle/graaljs. [Accessed 17 4 2021].

[7] Oracle Labs, "TruffleRuby," [Online]. Available: https://github.com/oracle/truffleruby. [Accessed 9 6 2021].

[8] Oracle Labs, "Graal Python," [Online]. Available: https://github.com/oracle/graalpython. [Accessed 9 6 2021].

[9] Oracle Labs, "GraalVM," [Online]. Available: https://www.graalvm.org/. [Accessed 12 4 2021].

[10] K. Hrbacek and T. Jech, "Elementary Operations on Sets," in *Introduction to Set Theory, Third Edition, Revised and Expanded*, New York, Marcel Dekker, 1999, pp. 12-16.

[11] T. Gjervig, "ZET," [Online]. Available: https://github.com/terkelg/zet. [Accessed 12 4 2021].

[12] Montage Studio, "collections.js," [Online]. Available: http://www.collectionsjs.com/. [Accessed 12 4 2021].

[13] Facebook, Inc, "immutable.js," [Online]. Available: https://immutable-js.github.io/immutable-js/. [Accessed 12 4 2021].

[14] ECMA International, "New Set Methods - Specification," [Online]. Available: https://tc39.es/proposal-set-methods/. [Accessed 5 4 2021].

[15] ECMA International, "ECMAScript 2015 Language Specification," 6 2015. [Online]. Available: https://262.ecma-international.org/6.0/. [Accessed 9 4 2021].

[16] Oracle Labs, "GraalVM Truffle Javadoc," [Online]. Available: https://www.graalvm.org/truffle/javadoc/. [Accessed 13 4 2021].

[17] Node.js Foundation, "Node.js," [Online]. Available: https://nodejs.org/en/. [Accessed 20 04 2021].

[18] Google-Inc, "V8," [Online]. Available: https://v8.dev/. [Accessed 20 04 2020].

[19] Netscape Communications, "SpiderMonkey," [Online]. Available: https://spidermonkey.dev/. [Accessed 20 04 2021].

[20] J. Küster, "Javascript Set Extensions," [Online]. Available: https://github.com/jankapunkt/js-set-extension. [Accessed 20 04 2021].

9 List of Listings

10 List of Tables

11 List of Figures