JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von

Christian Aistleitner

Angefertigt am

Institut für Systemsoftware

Betreuer

**Dipl. Ing.
Sebastian Kloibhofer**

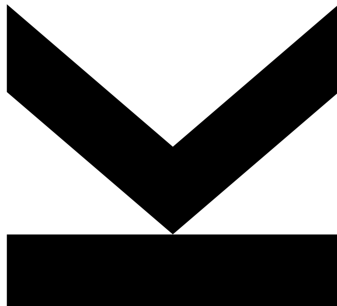
Mitbetreuer

**Dr. Christian Wirth
(Oracle Labs)**

August 2021

RECORD & TUPLE ECMASCRIPT PROPOSAL IN GRAAL.JS

Bachelor Thesis with Oracle Labs



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

Abstract

Graal.js [1] is an ECMAScript 2021 compliant JavaScript implementation by Oracle built on GraalVM [2]. It is based on the Truffle framework [3] and makes use of specialization to optimize its execution, e.g. specialization on the actually used data types or other types of runtime feedback.

The ECMAScript specification [4] is amended by so-called proposals. This Bachelor thesis focuses on the **Record & Tuple proposal** [5] authored by Robin Ricard (Bloomberg), Rick Button (Bloomberg), and Nicolò Ribaudo (Babel). Records and tuples are deeply immutable data structures. Data stored in those data types cannot be changed once created.

Part of this Bachelor thesis is an implementation of this proposal as part of the core Graal.js interpreter. The task comprises the full functionality of the current state of the proposal. In addition to the full functionality, the implementation is required to show a good performance. For this reason, optimizing techniques provided by the Truffle framework are used to meet this requirement.

This thesis presents the implementation of the proposal and describes the approach for verifying the correctness of the implementation. Furthermore, this thesis presents a performance evaluation of the added data types in comparison with similar cases in traditional JavaScript using custom benchmarks.

Kurzfassung

Graal.js [1] ist eine ECMAScript 2021 konforme JavaScript Implementierung von Oracle und basiert auf GraalVM [2] sowie dem Truffle-Framework [3]. Um die Ausführung zu optimieren, wird u.a. auf Spezialisierung gesetzt, z.B. Spezialisierung auf die tatsächlich verwendeten Datentypen oder mit Hilfe von sonstigem Laufzeit-Feedback.

Die ECMAScript-Spezifikation [4] wird durch sogenannte Proposals ergänzt. Diese Bachelorarbeit beschäftigt sich mit dem **Record & Tuple Proposal** [5] von Robin Ricard (Bloomberg), Rick Button (Bloomberg) und Nicolò Ribaudo (Babel). Die vom Proposal vorgeschlagenen neuen Datentypen sind spezielle Datenstrukturen, deren Inhalt nach ihrer Erstellung nicht mehr geändert werden können.

Teil dieser Bachelorarbeit ist eine Umsetzung dieses Proposals als Teil des Graal.js Interpreters. Die Aufgabe umfasst die volle Funktionalität zum aktuellen Stand des Proposals. Neben der vollen Funktionalität muss die Implementierung eine gute Performance aufweisen. Um diese Anforderung zu erfüllen, werden Optimierungstechniken des Truffle-Frameworks verwendet.

Diese Arbeit präsentiert die Umsetzung des Proposals und beschreibt den Ansatz zur Überprüfung auf Korrektheit. Darüber hinaus präsentiert diese Arbeit eine Performance-Auswertung der hinzugefügten Datentypen im Vergleich zu ähnlichen Szenarien in traditionellem JavaScript mit Hilfe von eigenen Benchmarks.

Contents

1 Introduction **4**

1.1 Problem & Motivation 4
1.2 Solution: Records and Tuple proposal 4
1.3 Contributions 5
1.4 Structure of this Thesis 5

2 Fundamentals **6**

2.1 ECMAScript 6
2.1.1 TC39 Process 7
2.2 GraalVM 8
2.2.1 Truffle 8
2.2.2 Graal.js 9
2.3 Records 10
2.4 Tuples 10

3 Record & Tuple Proposal **11**

3.1 Data Types 11
3.2 Syntax 11
3.3 Semantics 12
3.4 Equality 13
3.5 Built-in Methods 14

4 Implementation **16**

4.1 Overview 16
4.2 Compiler Frontend 17
4.3 Building the AST 20

5 Testing **23**

5.1 Existing Test Suites 23
5.2 Unit Tests 23

6 Performance Evaluation **25**

6.1 Benchmark 25
6.2 Setup 25
6.3 Results 26

7 Conclusion **27**

8 References **28**

1 Introduction

This chapter outlines the problem to solve and existing related work. Section 1.1 describes the problem and the need for a better solution. Section 1.2 presents an existing approach for solving this issue, while Section 1.3 clarifies the actual contributions of this thesis. Finally, Section 1.4 gives an overview of the upcoming chapters.

1.1 Problem & Motivation

JavaScript [6] (in short: *JS*) is a high-level and lightweight programming language. It's one of the core technologies of modern-day websites besides HTML and CSS. During execution, JS code is usually interpreted or just-in-time compiled.

Although JS has had many years to evolve after its first appearance in 1995, it doesn't yet feature immutable data structures. Other popular programming languages do support them, e.g. Python features an immutable data structure called `Tuple`.

Data stored in immutable data types cannot be changed (mutated) once created. This concept is very useful as it results in much simpler application development especially as projects grow larger. *Defensive Copying*, the copying of data in order to prevent changes, gets obsolete. Furthermore, the use of immutable data structures increases predictability and thus results in a more stable software product.

For this reason, userland libraries like *Immutable.js* [7] and *Immer* [8] were created to solve this issue. Although those libraries are doing a great job, a native solution would result in a much better programming and debugging experience. In addition, it is expected that a native implementation would improve execution performance.

1.2 Solution: Records and Tuple proposal

JavaScript is an implementation of the ECMAScript specification [4]. The ECMAScript specification is amended by so-called *proposals.* This Bachelor thesis focuses on the **Record & Tuple proposal** [5] (in short: *RTP*) authored by Robin Ricard (Bloomberg), Rick Button (Bloomberg), and Nicolò Ribaudo (Babel).

The proposal provides a possible solution for adding deeply immutable data structures to the ECMAScript language. Technical details are presented later on in Chapter 3.

1.3 Contributions

Graal.js is an ECMAScript 2021 compliant JavaScript implementation by Oracle built on GraalVM [2]. It is based on the Truffle framework [3] and makes use of specialization to optimize its execution, e.g. specialization on the actually used data types or other types of runtime feedback.

A goal of this Bachelor thesis is an open-source contribution to the Graal.js repository [1] under the Universal Permissive License[1] (in short: *UPL*). The contribution comprises an implementation of the RTP as part of the core Graal.js interpreter. The implementation covers the full functionality of the proposal at the time of starting this Bachelor thesis in February 2021.

1.4 Structure of this Thesis

After the introduction, Chapter 2 describes the fundamental technologies this work is based on. A technical description of the task is presented in Chapter 3, followed by the concrete implementation in Chapter 4. In Chapter 5, the approach for verifying the correctness of the implementation is described. Chapter 6 features a performance evaluation. Lastly, this thesis finishes with a conclusion in Chapter 7.

[1] https://opensource.org/licenses/UPL

2 Fundamentals

This chapter presents some fundamental concepts relevant to this thesis. In Section 2.1, ECMAScript as language and ecosystem is introduced. Section 2.2 describes GraalVM and its platform for executing guest languages. Finally in Section 2.3 and Section 2.4 basic theoretical concepts of records and tuples are outlined.

2.1 ECMAScript

ECMAScript [4] is a general-purpose, cross-platform, and vendor-neutral programming language. The development of the ECMAScript Language Specification [9] started in November 1996 and since the publication of its first edition in 1997, received many updates. At the time of writing, the latest version is the 12th edition also known as *ECMAScript 2021*, *ES2021* or *ES12*.

ECMAScript is based on several originating technologies, the most well-known being JavaScript (Netscape) and JScript (Microsoft). It was introduced as a Web scripting standard meant to ensure the interoperability of web pages across different web browsers [10]. For this reason, it is commonly used for client-side scripting and thus is best known as the language embedded in web browsers. However, it has also been widely adopted for server and embedded applications using for example Node.js [11] as JavaScript runtime.

The ECMAScript Language Specification [9] is also known as ECMA-262 and is maintained by Ecma International [12], an industry association dedicated to the standardization of information and communication systems. Technical work is carried out by Technical Committees (TCs) and Task Groups (TGs) [13]. A Technical Committee or a Task Group addresses a single area or topic.

The ECMA Technical Committee 39 [14] (in short: *TC39*) focuses on the the ECMAScript language which includes

- maintaining and updating the ECMAScript standards [9][15],
- developing standards for complementary technologies and libraries,
- developing test suites [16] which are used to verify implementations,
- and evaluating proposals [17].

2.1.1 TC39 Process

The TC39 Process [18] defines how changes in the specification are made. Proposals following this process are tracked in a public Git repository [17] hosted on GitHub.

There are 5 stages and the TC39 committee must approve before a proposal can move to the next stage.

- **Stage 0: Strawperson**
  The purpose of this stage is to allow input into the specification.
- **Stage 1: Proposal**
  In this stage, the initial idea is being taken to a formal proposal that describes not only the problem or need for the addition but also suggests potential solutions and challenges. A working demo or polyfill implementation is expected.
- **Stage 2: Draft**
  A draft requires an initial specification text containing all major syntax and semantic changes. Further incremental changes are to be expected in this stage.
- **Stage 3: Candidate**
  Proposals in this stage are almost final. Designated reviewers and all ECMAScript editors have signed off on the specification text and changes may only occur if something critical is being found while working on implementations.
- **Stage 4: Finished**
  The last stage indicates that the addition is ready for inclusion in the ECMAScript standard and will be included in the next upcoming edition. Acceptance tests have been written for mainline usage scenarios.

2.2 GraalVM

GraalVM is a high-performance JDK distribution based on the Java HotSpot VM from Oracle [19]. It uses the *GraalVM Compiler* as its dynamic just-in-time compiler, which transforms Java bytecode into the target architecture's machine code.

Figure 1 shows the connection between the Java HotSpot VM and the GraalVM Compiler. Additionally, it assigns some of their core components. As depicted, the compiler interacts with the JVM using the low-level JVM Compiler Interface (in short: *JVMCI*).

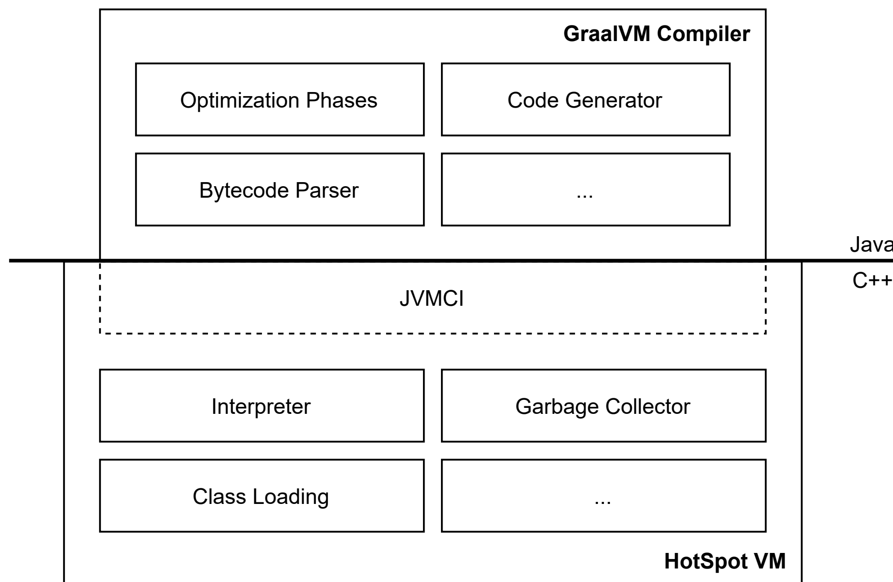


Figure 1: GraalVM interface architecture

By using multiple optimization algorithms [20] (also called "Phases"), like aggressive inlining or polymorphic inlining, the GraalVM Compiler produces highly optimized machine code. It is designed to accelerate the execution of applications written in Java and other JVM languages like Scala, Kotlin, or Groovy.

2.2.1 Truffle

Truffle [3] is an open-source language implementation framework. It provides building tools for guest language implementations and is built on top of the GraalVM as illustrated in Figure 2 below.

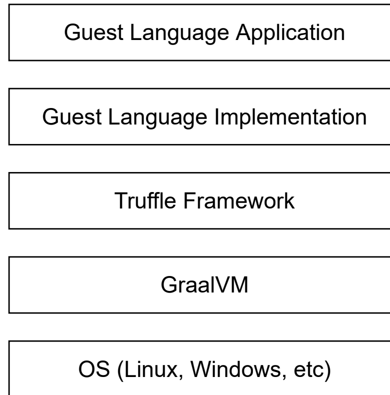


Figure 2: Truffle system architecture

Several guest-language implementations are provided already, for instance for JavaScript, Ruby, and Python. Additionally, Truffle provides polyglot capabilities using the GraalVM Polyglot API. This allows developers to embed source code written in any supported guest language into their JVM-based host applications.

Guest languages are implemented as *Abstract Syntax Trees* (in short: *AST*) interpreters [21]. A given program written in a guest language gets translated into an AST consisting of instances of Truffle nodes. A Truffle node is a subclass of `Node` and requires a special `execute` method which is being called when traversing the tree during the execution.

Truffle uses two optimization techniques for boosting the execution performance:

- **Specialization**
  Using type feedback and other profiling information, AST nodes specialize in order to fit best their given situation.
- **Partial Evaluation** (in short: *PE*)
  PE is the process of transforming a Truffle AST to highly optimized machine code, by inlining all nodes of one compilation unit (function) and using profiling feedback to optimize the result.

2.2.2 Graal.js

Graal.js [1], also known as *GraalVM JavaScript Implementation* [22], provides an ECMAScript-compliant runtime for executing JS and Node.js applications. It is built upon the Truffle framework and thus provides all benefits the GraalVM stack offers, including language interoperability and common tooling.

2.3 Records

A Record is a data structure that holds a collection of fields. Those fields are fixed in number and may have different data types. Listing 1 shows a typical example in Kotlin.

```
1 data class Person(
2     val firstname: String,
3     val lastname: String,
4 )
5
6 fun main() {
7     var p = Person("John", "Doe")
8     println("Hello " + p.firstname) // Output: Hello John
9 }
```

Listing 1: Record example in Kotlin

2.4 Tuples

A Tuple is a data structure that holds an ordered sequence of elements. Like most terms used in computer science, the term ”tuple” originates from mathematics. There, tuples serve a variety of purposes, but it is most commonly used to express sequences.

In computer science, a tuple is usually an immutable array-like data type. As already mentioned in Section 1.1, this immutable data structure is implemented in the Python programming language [23]. Listing 2 shows a code snippet showcasing Python’s `Tuple`.

```
1 cities = ("Linz", "Graz", "Vienna")
2 print("Welcome to " + cities[0]) # Output: Welcome to Linz
```

Listing 2: Tuple example in Python

3 Record & Tuple Proposal

This chapter describes a possible solution based on the RTP for adding two immutable data structures to the ECMAScript specification. In Section 3.1, the newly added data types are introduced. Section 3.2 outlines the syntax, while Section 3.3 describes the semantics of the proposed solution. Section 3.4 illustrates how ECMAScript's equality algorithms should handle records and tuples. In Section 3.5, a complete list of the proposed built-in methods is shown.

3.1 Data Types

The RTP adds two new deeply immutable data structures to the set of built-in types:

- **Record**
  A deeply immutable primitive type that contains mappings from Strings to primitive values. Note that in JS, all primitive values are immutable.
- **Tuple**
  A deeply immutable primitive type containing an ordered sequence of primitives.

3.2 Syntax

The RTP proposes a syntax similar to the existing solution for defining objects or arrays. By using the # modifier in front of the opening bracket, an otherwise normal object or array expression gets converted into a record or tuple expression. A syntax example can be seen in Listing 3.

```
1 const object = { id: 1, data: "Hello World!"};
2 const record = #{ id: 1, data: "Hello World!"};
3
4 const array = [1, 2, 3];
5 const tuple = #[1, 2, 3];
```

Listing 3: Syntax example

With the exception of arrays containing holes, any array expression can be converted to a tuple expression by adding the # modifier. Holes in a tuple expression, similar as shown in Listing 4, will raise a syntax error.

```
1 const x = #[,]; // SyntaxError: Unexpected token: ,
```

Listing 4: Holes are disallowed by syntax

There are also a few limitations when converting object expressions to record expressions from a syntax perspective. Records cannot contain methods nor getter or setter functions as shown in Listing 5. In addition, properties named `__proto__` are not allowed as record values do not participate in prototype chains.

```
const x = #{
    method() { }
}; // SyntaxError: Expected : but found (

const y = #{
    get id() { return 42; }
}; // SyntaxError: Expected : but found id

const z = #{
    __proto__: 42
}; // SyntaxError: '__proto__' is not allowed in Record expressions
```

Listing 5: Record syntax errors

Comparing this syntax with existing implementations in other languages, Python is using a similar approach. The biggest difference is in the different syntax, where tuples in Python do not start with a `#` modifier prefix as they are using a different type of brackets for distinguishing them from arrays. An example can be seen in Listing 6.

```
# python
array_value = [1, 2, 3]
tuple_value = (1, 2, 3)
```

Listing 6: Python tuple

3.3 Semantics

The RTP specifies that record and tuple values are *deeply* immutable. Therefore, defining records or tuples using mutable values will raise a `TypeError` as can be seen in Listing 7.

```
const obj = { id: 1 };

const x = #{ data: obj };
// TypeError: Records cannot contain non-primitive values

const y = #[1, 2, 3, obj];
// TypeError: Tuples cannot contain non-primitive values
```

Listing 7: TypeError due to non-primitive values

Although usually any value of type `String` or `Symbol` can be used as property key in ECMAScript, records will raise a `TypeError` if `Symbols` are being used.

```
const sym = Symbol('test');

const x = #{[sym]: 42};
// TypeError: Record may only have string as keys
```

Listing 8: TypeError due to symbol property key

3.4 Equality

Record and tuple values are considered equal if their structure and contents are deeply identical. Note that strict equality (using `===`), loose equality (using `==`) and the internal *SameValueZero* algorithm treat `+0` and `-0` within record or tuples as equal. However, the `Object.is` built-in method uses the internal *SameValue* algorithm that treats `+0` and `-0` as unequal and thus returns a different result.

```
console.log(
    #{ a: 0 } === #{ a: -0 } && #[0] === #[-0] // strict equality
); // Output: true

console.log(
    #{ a: 0 } == #{ a: -0 } && #[0] == #[-0] // loose equality
); // Output: true

console.log(
    Object.is(#{ a: 0 }, #{ a: -0 }) && Object.is(#[0], #[-0])
); // Output: false
```

Listing 9: Equality of records and tuples

3.5 Built-in Methods

Besides the two data types, the RTP adds built-in methods for running operations on them. This includes the following 7 constructor built-in methods:

- `Record(arg)`
  `Record.fromEntries(iterable)`
  Returns a new record value based on the given argument.
- `Record.isRecord(arg)`
  Returns `true` if `arg` is a record value or record object.
- `Tuple(...items))`
  `Tuple.of(...items)`
  `Tuple.from(items [, mapFn [, thisArg]])`
  Returns a new tuple value based on the given arguments.
- `Tuple.isTuple(arg)`
  Returns `true` if `arg` is a tuple value or tuple object.

In addition, the proposal also includes 32 prototype built-ins for tuples and currently none for records. Note that only the first few of those are explained as most of them are self-explanatory.

- `get Tuple.prototype.length`
  A getter function returning the length of the tuple.
- `Tuple.prototype.valueOf()`
  Returns `this` as tuple value.
- `Tuple.prototype.popped()`
  Returns a tuple containing the elements of the tuple, except for the last value.
- `Tuple.prototype.pushed(...args)`
  Returns a Tuple containing the elements of the tuple, followed by the arguments.
- `Tuple.prototype.reversed()`
  Returns a tuple containing the elements of the tuple, in reverse order.
- `Tuple.prototype.shifted()`
  Returns a tuple containing the elements of the tuple, except for the first value.
- `Tuple.prototype.slice(start, end)`
  Returns a tuple containing the elements of the tuple from index `start` to `end-1`.
- `Tuple.prototype.sorted(comparefn)`
  Returns a tuple containing a sorted sequence of the elements of the tuple.

- `Tuple.prototype.spliced(start, deleteCount, ...items)`
- `Tuple.prototype.concat(...args)`
- `Tuple.prototype.includes(searchElement [, fromIndex])`
- `Tuple.prototype.indexOf(searchElement [, fromIndex])`
- `Tuple.prototype.join(separator)`
- `Tuple.prototype.lastIndexOf(searchElement [, fromIndex])`
- `Tuple.prototype.entries()`
- `Tuple.prototype.every(callbackfn [, thisArg])`
- `Tuple.prototype.filter(callbackfn [, thisArg])`
- `Tuple.prototype.find(predicate [, thisArg])`
- `Tuple.prototype.findIndex(predicate [, thisArg])`
- `Tuple.prototype.flat([depth])`
- `Tuple.prototype.flatMap(mapperFunction [, thisArg])`
- `Tuple.prototype.forEach(callbackfn [, thisArg])`
- `Tuple.prototype.keys()`
- `Tuple.prototype.map(callbackfn [, thisArg])`
- `Tuple.prototype.reduce(callbackfn [, thisArg])`
- `Tuple.prototype.reduceRight(callbackfn [, thisArg])`
- `Tuple.prototype.some(callbackfn [, thisArg])`
- `Tuple.prototype.unshifted(...args)`
- `Tuple.prototype.toLocaleString([reserved1 [, reserved2]])`
- `Tuple.prototype.toString()`
- `Tuple.prototype.values()`
- `Tuple.prototype.with(index, value)`

4 Implementation

This chapter describes the actual implementation of the RTP in Graal.js. In Section 4.1, an overview of the relevant phases is introduced. In Section 4.2, the compiler frontend, including the necessary parser changes, is presented. Section 6.3. describes the relevant AST nodes, while Section 6.4. is presenting the Truffle nodes in detail.

4.1 Overview

The process of executing JS code using Graal.js can be split into three phases as can be seen in Figure 3.

- **Compiler Frontend**
  Reads the JS code and produces the intermediate representation (in short: *IR*).
- **Translator**
  Translates the IR produced by the compiler frontend into AST nodes.
- **AST**
  The final result which gets executed.

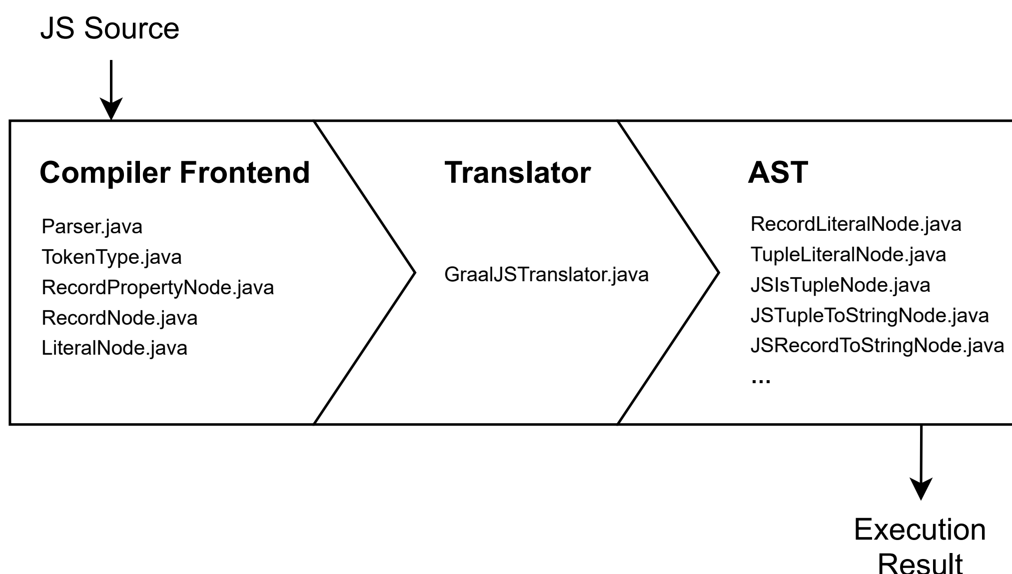


Figure 3: Implementation architecture

The following sections in this chapter are about the necessary changes for supporting records and tuples as described in Chapter 5.

4.2 Compiler Frontend

The compiler frontend consists of two main parts:

- **Lexer:** Responsible for converting source content into a stream of tokens.
- **Parser:** A recursive descent parser which builds the IR.

In order to implement the proposed JS grammar changes as described in Chapter 5, the existing code needs to be extended.

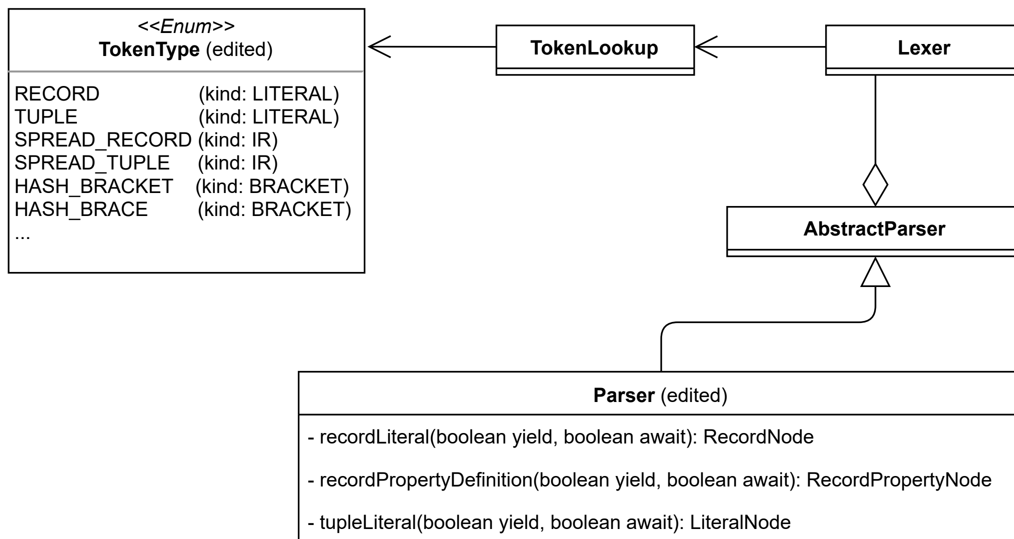


Figure 4: Compiler Frontend UML

By adding `HASH_BRACKET` and `HASH_BRACE` to the TokenType enum, as can be seen in Listing 10 below, the Lexer gets aware of those new keywords. There weren't any changes necessary in the Lexer class itself.

```java
public enum TokenType {
    ...
    HASH_BRACKET   (BRACKET, "#[", 0, true, 13),
    HASH_BRACE     (BRACKET, "#{", 0, true, 13);
}
```

Listing 10: TokenType.java

New methods for the nonterminal symbols (in short: *NTS*) `RecordLiteral`, `RecordPropertyDefinition` and `TupleLiteral` have been added to the recursive descent parser.

The implementation of the `TupleLiteral` NTS-method one can be seen in Listing 11 below. It is based on the existing code for parsing `Arrays`.

```
1  private LiteralNode tupleLiteral(boolean yield, boolean await) {
2      final long tupleToken = token; // Capture HASH_BRACKET token.
3      next(); // HASH_BRACKET tested in caller.
4  
5      // Prepare to accumulate elements.
6      final ArrayList<Expression> elements = new ArrayList<>();
7      // Track elisions.
8      boolean elision = true;
9      loop: while (true) {
10         long spreadToken = 0;
11         switch (type) {
12             case RBRACKET:
13                 next();
14                 break loop;
15 
16             case COMMARIGHT:
17                 if (elision) { // If no prior expression
18                     throw error(AbstractParser.message("unexpected.token",
       type.getNameOrType()));
19                 }
20                 next();
21                 elision = true;
22                 break;
23 
24             case ELLIPSIS:
25                 spreadToken = token;
26                 next();
27                 // fall through
28 
29             default:
30                 if (!elision) {
31                     throw error(AbstractParser.message("expected.comma",
       type.getNameOrType()));
32                 }
33 
34                 // Add expression element.
35                 Expression expression = assignmentExpression(true, yield,
       await, false);
36                 if (expression != null) {
37                     if (spreadToken != 0) {
```

```
38                         expression = new UnaryNode(Token.recast(spreadToken
    , SPREAD_TUPLE), expression);
39                     }
40                     elements.add(expression);
41                 } else {
42                     expect(RBRACKET);
43                 }
44 
45                 elision = false;
46                 break;
47         }
48     }
49 
50     return LiteralNode.newTupleInstance(tupleToken, finish, optimizeList(
    elements));
51 }
```

Listing 11: tupleLiteral(...) in Parser.java

The added NTS-methods for parsing `Records` and `Tuples` are being called in the NTS-method for `PrimaryExpression` as can be seen in Listing 12. If the current token type equals the start of a record literal (`#{`), the parser will execute `recordLiteral(...)`. In case it equals the start of a tuple literal (`#[`), it will call `tupleLiteral(...)`.

```
1  private Expression primaryExpression(boolean yield, boolean await) {
2      // Capture first token.
3      final int primaryLine = line;
4      final long primaryToken = token;
5  
6      switch (type) {
7          ...
8          case LBRACKET:
9              return arrayLiteral(yield, await);
10         case LBRACE:
11             return objectLiteral(yield, await);
12         case HASH_BRACE:
13             return recordLiteral(yield, await);
14         case HASH_BRACKET:
15             return tupleLiteral(yield, await);
16         ...
17     }
18     ...
19 }
```

Listing 12: primaryExpression(...) in Parser.java

4.3 Building the AST

After parsing the JS file, the literal nodes created by the parser need to be translated into executable AST nodes. An overview of the affected Node classes and how they are accessed from `GraalJSTranslator` can be seen in Figure 5. The `GraalJSTranslator` uses the visitor design pattern to do this translation. This section focuses on the translation of tuples. The process is very similar for record literals.

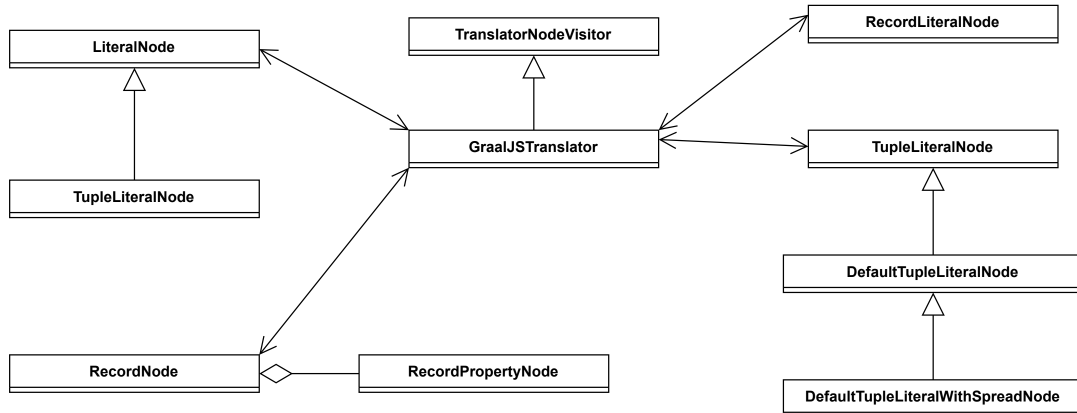


Figure 5: Translator UML

Listing 13 shows the `GraalJSTranslator` method which gets called when a `LiteralNode` containing a tuple is visited. It checks if the tuple contains a spread operator (`...foo`) as in this case, a special subclass of `TupleLiteralNode` gets created.

```
private JavaScriptNode enterLiteralTupleNode(LiteralNode.TupleLiteralNode tupleLiteralNode) {
    List<Expression> elementExpressions = tupleLiteralNode.getElementExpressions();
    JavaScriptNode[] elements = javaScriptNodeArray(elementExpressions.size());
    boolean hasSpread = false;
    for (int i = 0; i < elementExpressions.size(); i++) {
        Expression elementExpression = elementExpressions.get(i);
        hasSpread = hasSpread || elementExpression.isTokenType(TokenType.SPREAD_TUPLE);
        elements[i] = transform(elementExpression);
    }
    return hasSpread ? factory.createTupleLiteralWithSpread(context, elements) : factory.createTupleLiteral(context, elements);
}
```

Listing 13: enterLiteralTupleNode(...) in GraalJSTranslator.java

`factory` in the listing above is an instance of `NodeFactory` which follows the proxy pattern and forwards the call to the corresponding AST node as can be seen in Listing 14.

```
public JavaScriptNode createTupleLiteral(JSContext context, JavaScriptNode
    [] elements) {
    return TupleLiteralNode.create(context, elements);
}

public JavaScriptNode createTupleLiteralWithSpread(JSContext context,
    JavaScriptNode[] elements) {
    return TupleLiteralNode.createWithSpread(context, elements);
}
```

Listing 14: createTupleLiteral(...), createTupleLiteralWithSpread(...) in NodeFactory.java

Finally, the AST node gets created. If possible, the actual tuple value gets initialized during the building of the AST as shown in Listing 15. The `create`-method returns a predefined empty tuple in case the `elements` array is empty or an AST node containing the final tuple value in case all contained elements are constants.

```
public abstract class TupleLiteralNode extends JavaScriptNode {
    ...

    public static JavaScriptNode create(JSContext context, JavaScriptNode[]
     elements) {
        if (elements == null || elements.length == 0) {
            return createEmptyTuple();
        }

        Object[] constantValues = resolveConstants(elements);
        if (constantValues != null) {
            return createConstantTuple(constantValues);
        }

        return new DefaultTupleLiteralNode(context, elements);
    }

    public static TupleLiteralNode createWithSpread(JSContext context,
    JavaScriptNode[] elements) {
        return new DefaultTupleLiteralWithSpreadNode(context, elements);
    }

    ...
}
```

Listing 15: create(...), createWithSpread(...) in TupleLiteralNode.java

In case the elements have to be resolved during execution, a `DefaultTupleLiteralNode` or `DefaultTupleLiteralWithSpreadNode` gets created. A reason for this can be the usage of the spread operator (`...data`) or computed values.

Listing 16 shows how we resolve computed values. Note that the `requireNonObject`-method checks the type of the computed value and throws a `TypeError` if the value is of type `object` as described in Section 3.3.

```
public abstract class TupleLiteralNode extends JavaScriptNode {
    ...

    private static class DefaultTupleLiteralNode extends TupleLiteralNode {

        @Children protected final JavaScriptNode[] elements;

        private DefaultTupleLiteralNode(JSContext context, JavaScriptNode[]
 elements) {
            super(context);
            this.elements = elements;
        }

        @Override
        public Tuple execute(VirtualFrame frame) {
            Object[] values = new Object[elements.length];
            for (int i = 0; i < elements.length; i++) {
                Object value = elements[i].execute(frame);
                values[i] = requireNonObject(value);
            }
            return createTuple(values);
        }

        ...
    }

    ...
}
```

Listing 16: DefaultTupleLiteralNode in TupleLiteralNode.java

5 Testing

This chapter is about verifying the correctness of the RTP implementation. In Section 5.1, we take a look at existing test suites, while Section 5.2 presents the implementation of custom unit tests.

5.1 Existing Test Suites

Test262 [16] is a test suite provided by TC39 to verify ES implementations. As the RTP is still in stage 2 and acceptance tests aren't required before stage 4, the repository doesn't contain tests testing the functionality of records and tuples. However, the implementation was tested against the existing Test262 tests in order to ensure that no regressions occurred.

Also TestV8, Google's test suite used for verifying the correctness of their V8 JS engine, doesn't contain test cases testing the functionality introduced by the RTP. For this reason, it was necessary to write custom tests as described in the next section.

5.2 Unit Tests

As Test262 and TestV8 don't contain the necessary tests for verifying the correctness of the RTP implementation, custom tests in the shape of unit tests were added.

Tests were introduced for every built-in method or property and for common operations such as equality checks or type conversions. Listing 17 shows one such unit test in which `testLength()` verifies that the built-in property `length` is set to the length of the tuple.

```java
public class TuplePrototypeBuiltinsTest extends JSSimpleTest {

    public TuplePrototypeBuiltinsTest() { ... }

    @Test
    public void testLength() {
        assertEquals(0, execute("#[].length").asInt());
        assertEquals(3, execute("#[1, 2, 3].length").asInt());
        assertEquals(3, execute("Object(#[1, 2, 3]).length").asInt());
    }

    ...
}
```

Listing 17: testLength() in TuplePrototypeBuiltinsTest.java

Unit test classes inherit the base class `JSSimpleTest` shown in Listing 18. It provides methods for executing JS snippets using the Polyglot API introduced in Section 2.2.1.

```java
public abstract class JSSimpleTest {

    protected final String testName;

    private final Map<String, String> options = new HashMap<>();

    protected JSSimpleTest(String testName) {
        this.testName = testName;
    }

    protected void addOption(String key, String value) {
        options.put(key, value);
    }

    protected Value execute(String sourceText) {
        try (Context context = newContext()) {
            return context.eval(Source.newBuilder(JavaScriptLanguage.ID,
    sourceText, testName).buildLiteral());
        }
    }

    protected Value execute(String... sourceText) {
        return execute(String.join("\n", sourceText));
    }

    protected void expectError(String sourceText, String expectedMessage) {
        try (Context context = newContext()) {
            context.eval(Source.newBuilder(JavaScriptLanguage.ID,
    sourceText, testName).buildLiteral());
            Assert.fail("should have thrown");
        } catch (Exception ex) {
            Assert.assertTrue(ex.getMessage().contains(expectedMessage));
        }
    }

    private Context newContext() {
        return Context.newBuilder(JavaScriptLanguage.ID)
                .allowExperimentalOptions(true)
                .options(options)
                .build();
    }
}
```

Listing 18: JSSimpleTest.java

6 Performance Evaluation

This chapter provides a performance evaluation of the added deeply immutable data types in comparison to similar existing data types. Benchmark and methodology details are described in section 6.1. The setup used for benchmarking is described in section 6.2. In section 6.3, the obtained results are presented.

6.1 Benchmark

Benchmark.js [24] was used as benchmark harness to avoid common pitfalls. The created custom benchmark suit tests two basic test scenarios:

- **Create**
  Creating a value of the given data type,
  e.g. `return #{ id: 1, data: "Hello World!" };`
- **Get**
  Accessing an element of a previously defined value of the given data type,
  e.g. `return tuple[1]; // somewhere outside: var tuple = #[1,2,3];`

Besides benchmarking the added `Record` and `Tuple` data types, existing similar implementations are also being tested for comparisons. This includes the JS-native `Object` and `Array` as well as `Map` and `List` provided by the Immutable.js [7] library.

After running 10M warmup iterations, the benchmarking library calculates the number of iterations required to reduce the margin of error (in short: *MOE*) to about 1% and starts executing the benchmark.

6.2 Setup

The tests were conducted on a desktop workstation running a clean install of Ubuntu Server 21.04 installed on a secondary internal SSD. The hardware specifications are:

- **CPU:** i7-10700 8C16T @2.90GHz
- **RAM:** 32GiB (2x16) DDR4 @2667MHz
- **SSD:** 1TB SATA

By running Ubuntu Server, external influences such as background system updates (e.g. Windows updates) are being reduced. The only additional packages installed are those required for building and running Graal.js.

6.3 Results

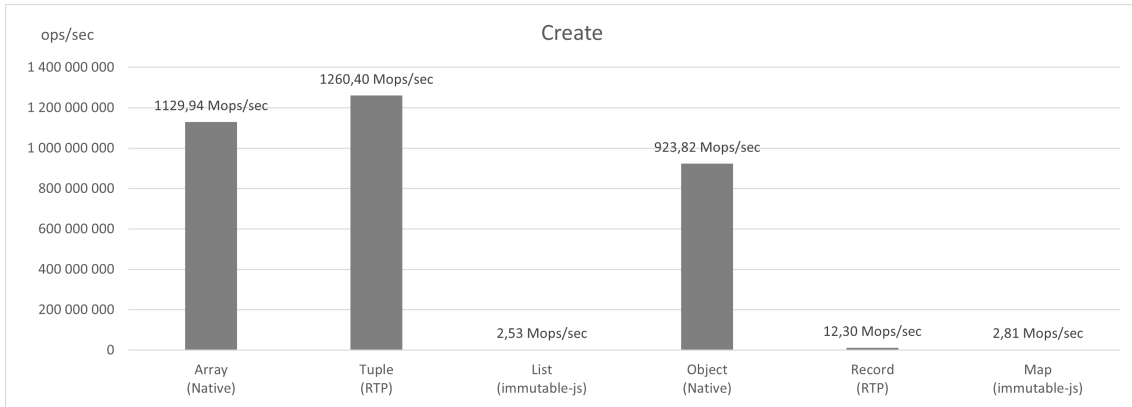


Figure 6: Benchmark 1, Create

Figure 6 illustrates the results of the first test scenario. Interestingly, the `Tuple` implementation outperforms the existing `Array` implementation by 11.5%. A potential reason for this are type checks. Graal.js's `Array` implementation performs a lot of type checks and stores the elements in a typed Java array if possible. Elements stored in `Tuple`s on the other hand are being stored in untyped Java object arrays and thus type checks aren't necessary. The low score of `List` and `Map` compared to `Array` and `Object` shows the performance improvement potential the RTP can offer.

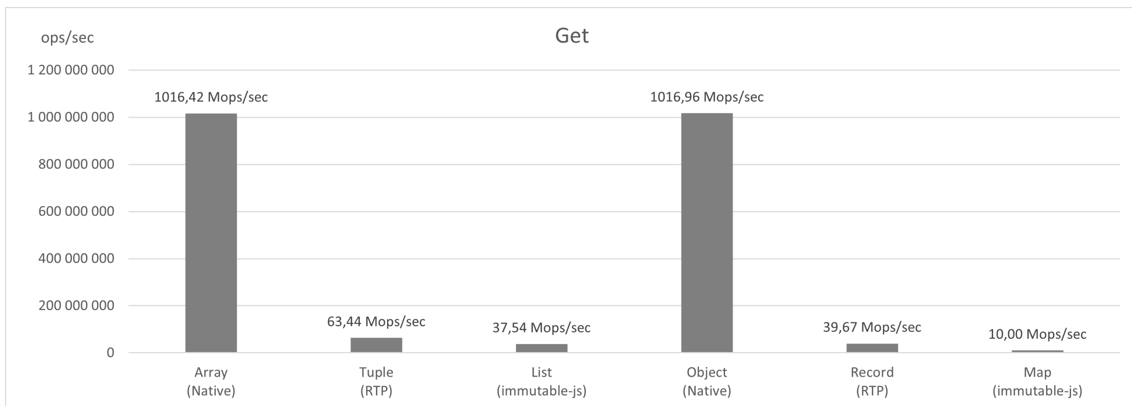


Figure 7: Benchmark 2, Get

Figure 7 shows our second test scenario. As values are being defined once and then usually accessed multiple times, having a high score in this test scenario is important. Although the added types outperform those provided by Benchmark.js by 69% and 297%, further improvements should be possible to archive results similar to `Array` or `Object`.

7 Conclusion

The main goal of this Bachelor thesis was an open-source contribution to the Graal.js repository [1]. The contribution comprised the full functionality of the RTP as of February 2021 as part of the core Graal.js interpreter.
The Graal.js pull request [25], which adds the RTP implementation, consists of a total of 76 changed or added files in 33 commits. It lists 7244 added lines and 15 removed ones.

The second goal of showing a good execution performance was also met. In one scenario, the `Tuple` implementation even outperformed the existing `Array` by 11.5%. Compared to the transitional way of using an userland library like Immutable.js [7], the RTP implementation showed an distinct performance improvement. However, further improvements are necessary to archive results similar to existing `Array` or `Object` implementations.

Additionally, there is future work to be done on the RTP implementation as the proposal is still expected to be changed and improved before it moves to stage 4 eventually.

8 References

List of Figures

1 GraalVM interface architecture . . . 8
2 Truffle system architecture . . . 9
3 Implementation architecture . . . 16
4 Compiler Frontend UML . . . 17
5 Translator UML . . . 20
6 Benchmark 1, Create . . . 26
7 Benchmark 2, Get . . . 26

List of Listings

1 Record example in Kotlin . . . 10
2 Tuple example in Python . . . 10
3 Syntax example . . . 11
4 Holes are disallowed by syntax . . . 11
5 Record syntax errors . . . 12
6 Python tuple . . . 12
7 TypeError due to non-primitive values . . . 12
8 TypeError due to symbol property key . . . 13
9 Equality of records and tuples . . . 13
10 TokenType.java . . . 17
11 tupleLiteral(...) in Parser.java . . . 18
12 primaryExpression(...) in Parser.java . . . 19
13 enterLiteralTupleNode(...) in GraalJSTranslator.java . . . 20
14 createTupleLiteral(...), createTupleLiteralWithSpread(...) in NodeFactory.java 21
15 create(...), createWithSpread(...) in TupleLiteralNode.java . . . 21
16 DefaultTupleLiteralNode in TupleLiteralNode.java . . . 22
17 testLength() in TuplePrototypeBuiltinsTest.java . . . 23
18 JSSimpleTest.java . . . 24

References

[1] Graal.js repository. URL: https://github.com/oracle/graaljs (visited on 2021-07-05).

[2] GraalVM. URL: https://www.graalvm.org/ (visited on 2021-07-23).

[3] Truffle Language Implementation Framework. URL: https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/ (visited on 2021-07-05).

[4] ECMA-262. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/ (visited on 2021-07-23).

[5] Record & Tuple Proposal Repository. URL: https://github.com/tc39/proposal-record-tuple (visited on 2021-07-07).

[6] JavaScript — MDN. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 2021-07-03).

[7] Immutable.js. URL: https://github.com/immutable-js/immutable-js/ (visited on 2021-07-06).

[8] Immer. URL: https://github.com/immerjs/immer (visited on 2021-07-06).

[9] ECMAScript® Language Specification. URL: https://tc39.es/ecma262/ (visited on 2021-07-03).

[10] A. Wirfs-Brock and B. Eich. JavaScript: The First 20 Years. *Proc. ACM Program. Lang.*, 4(HOPL), 2020-06. DOI: 10.1145/3386327. URL: https://doi.org/10.1145/3386327.

[11] Node.js. URL: https://nodejs.org/en/ (visited on 2021-07-03).

[12] Home - Ecma International. URL: https://www.ecma-international.org/ (visited on 2021-07-02).

[13] Technical Committees and Task Groups - Ecma International. URL: https://www.ecma-international.org/technical-committees/ (visited on 2021-07-02).

[14] TC39 - Ecma International. URL: https://www.ecma-international.org/technical-committees/tc39/ (visited on 2021-07-02).

[15] ECMAScript® Internationalization API Specification. URL: https://tc39.es/ecma402/ (visited on 2021-07-25).

[16] Test262 Repository. URL: https://github.com/tc39/test262 (visited on 2021-07-21).

[17] tc39/proposals: Tracking ECMAScript Proposals. URL: https://github.com/tc39/proposals (visited on 2021-07-04).

[18] The TC39 Process. URL: https://tc39.es/process-document/ (visited on 2021-07-04).

[19] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, Indianapolis, Indiana, USA. Association for Computing Machinery, 2013. ISBN: 9781450324724. DOI: 10.1145/2509578.2509581. URL: https://doi-1org-1007dacxb0519.han.ubl.jku.at/10.1145/2509578.2509581.

[20] GraalVM Compiler. URL: https://www.graalvm.org/reference-manual/compiler/ (visited on 2021-07-05).

[21] C. Wimmer and T. Würthinger. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, Tucson, Arizona, USA. Association for Computing Machinery, 2012. ISBN: 9781450315630. DOI: 10.1145/2384716.2384723. URL: https://doi-1org-1007dacxb057a.han.ubl.jku.at/10.1145/2384716.2384723.

[22] GraalVM JavaScript Implementation. URL: https://www.graalvm.org/reference-manual/js/ (visited on 2021-07-05).

[23] The Python Language Reference. URL: https://docs.python.org/3/reference/ (visited on 2021-07-26).

[24] Benchmark.js. URL: https://benchmarkjs.com/ (visited on 2021-07-19).

[25] Pull Request: Records & Tuples Proposal Implementation. URL: https://github.com/oracle/graaljs/pull/433 (visited on 2021-07-26).