TÉCNICO LISBOA

Infra-estrutura de Testes para Implementações de Referência do Standard ECMAScript

Miguel Maria Marçalo Pires Trigo

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Prof. José Faustino Fragoso Femenin dos Santos
Prof. António José dos Reis Morgado

Júri

Presidente: Prof. Name of the Chairperson
Orientador: Prof. José Faustino Fragoso Femenin dos Santos
Vogais: Prof. Name of First Committee Member
Dr. Name of Second Committee Member
Eng. Name of Third Committee Member

Month 20XX

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Abstract

JavaScript is one of the most used programming languages in the world. The specification of JavaScript is described in the ECMAScript standard, which consists in a complex and extensive document. An implementation of the ECMAScript standard faces many challenges in its testing process. Our thesis proposes an implementation of an infrastructure that mains to facilitate the testing process of the ECMAScript implementations. Our infrastructure uses two NoSQL databases: one database that stores the metadata of the Test262, the official test suite of JavaScript, tests; and another database that stores the results of the execution of the tests. The infrastructure provides a filter mechanism by feature and a process of automating the testing mechanism.

Keywords

ECMAScript, Specification Language, Reference Interpreters, Dynamic Languages, Test262, OCaml.

Resumo

JavaScript é uma das linguagens de programação mais utilizadas no mundo. A sua especificação é descrita no standard ECMAScript, que consiste num documento complexo e extenso. Uma implementação de referência do standard ECMAScript enfrenta vários desafios no processo de teste. A nossa tese propõe a implementação de uma infra-estrutura que facilitará o processo de teste de implementações de referência do standard ECMAScript. A nossa infra-estrutura utiliza bases de dados NoSQL: uma base de dados para guardar os metadados dos testes da batéria oficial de testes, Test262, e a sua localização; outra base de dados para armazenar os resultados obtidos da execução dos testes. A infra-estrutura fornece um mecanismo de filtragem dos testes por feature e um processo de automação do mecanismo de testes. A tese também inclui o desenvolvimento de uma implementação de referência das secções core do standard ECMAScript 6. Esta implementação é testada utilizando a infra-estrutura desenvolvida no projeto tese.

Palavras Chave

ECMAScript, Specification Language, Reference Interpreters, Dynamic Languages, Test262, OCaml

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

JavaScript (JS) é a linguagem de programação padrão para desenvolvimento de aplicações web do lado cliente, para além de ser uma das linguagens de programação mais utilizadas em geral. De acordo com os dados estatísticos do Github e do StackOverflow, a linguagem JavaScript é a mais ativa no Github[1] e a segunda mais ativa no StackOverflow[2]. A especificação da linguagem JavaScript é descrita no standard EcmaScript [1], que consiste num documento complexo e extenso. JavaScript é difícil de aprender e de analisar estaticamente devido à sua complexidade e constante crescimento. Este crescimento é visível na dimensão da especificação da linguagem, como pode ser verificado na Figura 1.1.

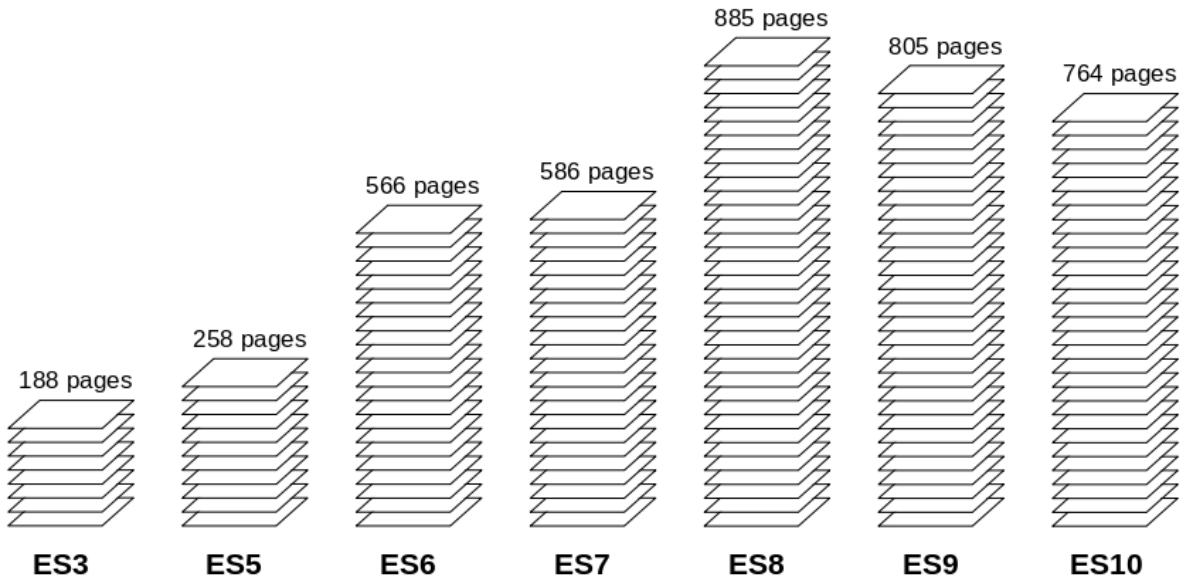

Figura 1.1: Evolução do número de páginas do documento oficial do standard ECMAScript

Como a maioria das linguagens dinâmicas, a linguagem JavaScript é normalmente interpretada. De entre os interpretadores mais relevantes destacam-se: o SpiderMonkey [2], o V8 [3] e o chakra [4]. Estes interpretadores fazem uso de representações intermédias e empregam a técnica just-in-time compilation para efeitos de performance. Existem também vários compiladores puramente estáticos para JavaScript (compiladores *ahead-of-time*), de entre os quais se destacam o Hop JavaScript compiler [5], o JSC [6] e o echoJS [7]. Devido ao carácter dinâmico da linguagem, os compiladores puramente estáticos são menos eficientes que os interpretadores baseados em just-in-time compilation.

A complexidade da linguagem JavaScript torna-a particularmente difícil de implementar uma vez que a semântica da linguagem exibe muitos comportamentos limite, usualmente designados por *corner-cases*. Por forma a testar implementações da linguagem é necessário criar baterias de testes que cubram todos os possíveis comportamentos, incluindo todos os *corner-cases*. Assim, durante o de-

[1] Linguagens de programação mais utilizadas no Github baseado em pull request - https://madnight.github.io/githut/
[2] Tendências no Stack Overflow baseado no uso das Tags - https://insights.stackoverflow.com/trends

senvolvimento de implementações da linguagem destacam-se três desafios relativos ao processo de teste:

1. *Como garantir que a bateria de testes utilizada cobre todos os comportamentos, em particular os corner cases da linguagem?*
   Mesmo a Test262 [8], a bateria de testes oficial do JavaScript, é reconhecidamente incompleta. Os criadores da Test262 afirmam que esta não é uma *conformance test suite*, indicando claramente que o facto de uma implementação da linguagem passar todos os teste não garante que esta esteja em total acordo com o standard. De facto, vários papers com implementações de referência do JavaScript têm identificado comportamentos limite por testar e contribuído com novos testes para a Test262 (ver Trabalhos Relacionados para mais informações).

2. *Como obter os testes apropriados para uma implementação parcial da linguagem?*
   Durante o desenvolvimento de uma nova implementação da linguagem é essencial testar as features implementadas à medida que estas são introduzidas. Isto gera a necessidade de filtrar a bateria de testes por feature. Por exemplo, se uma dada implementação da linguagem não suporta expressões regulares, deve ser possível excluir facilmente todos os testes que usam expressões regulares, mesmo aqueles que não testam explicitamente funcionalidades relacionadas com expressões regulares.

3. *Como gerir o processo de desenvolvimento de forma a garantir que o número de testes que a implementação da linguagem passa é crescente?*
   O processo de desenvolvimento deve incluir testes regulares, isto é periodicamente correr testes que confirmem que o processo feito até ao momento está bem implementado. Sempre que uma nova feature da linguagem é implementada devemos garantir que todos os testes que passavam anteriormente continuam a passar e que os testes correspondentes à nova feature implementada agora também passam. Para facilitar este processo, uma infra-estrutura para desenvolvimento de implementações de referência deve manter um registo sobre a cobertura da implementação atual e sobre o resultado da execução dos testes (testes executados com sucesso, testes falhados com anotações sobre a razão da falha). Assim, sempre que se implementar uma nova feature pode facilmente determinar-se quais dos testes que até então falhavam passaram a passar e vice-versa.

Nesta tese construímos uma infra-estrutura que auxilie o desenvolvimento de implementações de referência da linguagem JavaScript. A infra-estrutura desenvolvida tem no seu centro duas bases de dados:

1. Uma base de dados de testes construída a partir da testsuite oficial. Esta base de dados guarda a localização dos testes e os metadados dos mesmos. O cálculo dos metadados é uma contribuição

essencial desta tese, porque até ao momento os testes da Test262 [8] não têm a si associados metadados formais; os poucos metadados previamente existentes estão inseridos em comentário nos próprios testes. Estes metadados permitem a filtragem dos testes por funcionalidade.

2. Uma base de dados de resultados para guardar o resultado da execução dos testes. Esta base de dados serve para armazenar os resultados dos testes executados e comparar os resultados da versão corrente com os resultados de versões anteriores. Esta base de dados permite-nos descobrir facilmente quais os testes que agora estão a passar e antes não passavam, bem como os testes que agora estão a falhar quando antes passavam.

A base de dados de testes contem os metadados associados aos testes da Test262 [8]. Uma parte essencial desta tese é a definição de um formato canónico para representação dos metadados dos testes, que abrange os metadados já existentes, bem como metadados adicionais. A definição dos metadados adicionais a incluir é informada pela experiência de implementação do standard de alguns colegas que se encontram a realizar implementações do standard: à medida que houve necessidade de novos metadados, os mesmos são calculados e adicionados à base de dados. Os metadados considerados e adicionados são: as construções sintáticas que ocorrem nos testes (número de ifs, whiles,..), a versão do standard, e as bibliotecas built-ins utilizadas pelos testes.

As bases de dados acima referidas são desenvolvidas em MongoDB [9]. MongoDB é uma base de dados que utiliza documentos JSON para armazenamento de dados, sendo classificada como um base de dados NoSQL. As bases de dados NoSQL diferenciam-se das bases de dados relacionais pelo facto de não modularem os dados armazenados através de representações tabulares. Estas bases de dados têm diversas vantagens sobre os métodos mais comuns usados nas bases de dados relacionais, nomeadamente:

1. Esquemas de documentos flexíveis: o armazenamento de dados em ficheiros JSON permite que quase qualquer estrutura de dados seja guardada e manipulada facilmente.

2. Acesso aos dados orientado para o código: o formato JSON facilita o acesso programático aos dados, dispensando a utilização de *wrappers* para conversão entre o formato de armazenamento e o formato de *runtime*, como verificado nas bases de dados relacionais.

3. Design favorável à mudança: as bases de dados NoSQL possibilitam a alteração da estrutura das bases de dados sem a necessidade de destruir a estrutura existente.

A tese estrutura-se como se descreve em seguida. Na Secção 2 falamos dos trabalhos relacionados com o tema deste projeto, enquanto que na Secção 3 descrevemos a metodologia e o design utilizados na implementação do processo de filtragem. Na Secção 4 apresentamos uma caracterização dos testes presentes na bateria de testes *Test262*. A secção seguinte, a Secção 5, descreve o sistema de apoio

à testagem em maior detalhe. A Secção 6 avalia os resultados do trabalho realizado no decorrer deste projeto. Por fim, a Secção 7 termina a tese, apresentando as principais conclusões e um sumário da proposta.

2

Trabalho Relacionado

Estudos científicos sobre análise de programas e técnicas de instrumentação para o JavaScript são variados e cobrem um grande leque de especificações como: sistemas de tipos ([10], [11], [12], [13], [14], [15], [16]), interpretadores abstratos ([17], [18], [19], [20]), análises *points-to* ([21]), lógicas de Hoare ([22], [23]), semântica operacional ([24], [25], [26]), representações intermédias e compiladores ([27]). Em seguida, apresentamos os trabalhos relacionados mais relevantes para esta tese desenvolvida, organizando-os numa linha temporal dividida nas Tabelas 2.1 e 2.2. Esta linha temporal foca-se nos marcos mais importantes no trabalho de investigação focado na formalização da linguagem JavaScript desde a sua criação em 1995.

1995	O JavaScript é lançado e implementado no Netscape Navigator 2.0 beta 3.
2005	Thiemann apresenta uma primeira tentativa de definição de um sistema de tipos para JavaScript [28]. O sistema rastreia as possíveis características de um objeto e sinaliza conversões de tipos suspeitas. O trabalho realizado não cobre aspetos vitais do JavaScript, tal como herança baseada em protótipos, e não foi acompanhado por uma implementação na altura da sua publicação.
2008	Maffeis et al. propõe uma primeira semântica operacional para a linguagem ECMAScript versão 3. (ES3) [29]. Esta semântica foi usada para analisar várias propriedades de segurança de aplicações web e mashups.
2010	Guha et al. [30] definem a linguagem λJS, um cálculo formal que captura as características mais fundamentais do ES3. λJS vem com um interpretador *Racket* [31] e um tradutor de programas ES3 em expressões λJS.
2012	Gardner et al. [32] adapta ideias da lógica de separação para fornecer uma lógica de Hoare escalável para um subconjunto de JavaScript, com base em uma semântica operacional fiel ao ES3.
2012	Politz et al. [33] extende a linguagem λJS do ES3 para o ES5 incluindo, pela primeira vez, uma semântica formal da JavaScript `property descriptors` e tratamento completo da declaração `eval`.
2014	Bodin et al. apresenta o *JSCert* [24], uma formalização do standard ES5 escrito em *Coq* [34], e *JSRef* um interpretador executável de referências extraido do *Coq* para o *OCaml* [35], provado correto em relação ao *JSCert* e testado usando os testes da bateria Test262 [8].

Tabela 2.1: Linha temporal que contém os trabalhos relacionados mais relevantes para os objetivos do projeto (1 of 2)

2015	Park et al. apresenta *KJS* [25], a semântica formal do ES5 mais completa até à data. O *KJS* foi desenvolvido usando a framework *K* [36], um sistema consolidado de re-escrita de termos que suporta vários tipos de análise simbólica baseados em *reachability logic* (All-Path Reachability Logic [37]). *KJS* foi testada cuidadosamente usando a bateria de testes, passando em todos os 2.782 testes do core da linguagem. O *KJS* pode ser executado simbolicamente, podendo ser usado para análise formal e verificação de programas ES5.
2015	Gardner et al. [38] estendem o projecto *JSCert* para suportar *arrays* ES5. Para tal, os autores ligam o *JSRef* à implementação da biblioteca Array do motor V8 [3] da Google. Os autores adicionalmente avaliam o estado atual do projeto *JSCert*, fornecendo uma análise detalhada da metodologia como um todo e uma análise detalhada dos testes que o projeto passa/falha.
2017	Fragoso Santos et al. cria a ferramenta *JaVert* [22]: uma ferramenta de verificação JavaScript que permite raciocínio semi-automático sobre as propriedades funcionais de correcção dos programas ES5. JaVert não assume qualquer simplificação da semântica da linguagem, analisando todos os *corner-cases* da linguagem. A carga de anotações necessária é substancial.
2018	Charguéraud et al. apresenta *JSExplain* [26], um interpretador de referências para a linguagem ES5 que segue as especificações e que produz traços de execução inspecionáveis. Também inclui uma interface de inspeção de código que permite ao programador executar passo a passo o seu programa ES5 mas também o próprio código do interpretador ES5, enquanto executa os programas ES5.
2020	Sampaio et al. [39] estendem a ferramenta JaVerT para dar suporte ao raciocínio automático sobre eventos em JavaScript. Para tal, os autores desenvolvem diretamente em JavaScript uma implementação de referência de JavaScript promises. Os autores usam-na para testar a sua implementação de referência, passando 344 testes do total de 474 testes dedicados a JavaScript promises. Cerca de 106 testes foram excluídos devido a features ES6 que ainda não são suportadas e a testes que requerem o modo *non-strict*.
2020	Jihyeok Park et al. apresenta o JISET [40] a primeira ferramenta que sintetiza automaticamente o analizador e que traduz AST-IR diretamente de uma especificação da linguagem dada.

Tabela 2.2: Linha temporal que contém os trabalhos relacionados mais relevantes para os objetivos do projeto (2 of 2)

3

Cálculo da Metadata

Conteúdo

Com esta tese propomos calcular metadata adiconal para os testes da bateria oficial de testes do JavaScript Test262. Utilizando, para além dos campos já existentes em alguns dos testes da *Test262*, três novos campos para descrever o teste que irão permitir uma melhor caracterização dos testes e posteriormente uma filtragem dos mesmos. Os novos campos para acrescentar à metadata dos testes são:

1. Construções sintáticas utilizadas no teste
2. Versão do standard a que o teste pertence
3. Built-ins chamados no teste

Nesta secção vamos começar por apresentar a metadata já presente nos testes da test262, ilustrando com um exemplo na Secção 3.1. Seguido do cálculo da nova metadata proposta a ser acrescentada aos testes que é especificado nas secções seguintes da tese. A Secção 3.2 apresenta o cálculo das construções sintáticas apresentando o método utilizado seguido de um exemplo. De seguida, a Secção 3.3 descreve o método desenvolvido para o cálculo da versão que recorre a três abordagens distintas. Finalizando, na Secção 3.4 com o cálculo dos built-ins com duas abordagens distintas. O maior foco foi destinado ao cálculo da versão do stantard a que o teste pertence, uma vez que é a metadata que consideramos de maior importância.

3.1 Metadata atual

Todos os testes da bateria de testes *test262* devem seguir uma estrutura específica, que é composta por três secções distintas:

1. Secção de direitos de autor: esta primeira secção é escrita em comentário, isto é, todas as linhas começam com a denotação de comentário `//`. O formato a utilizar deve ser o apresentado na Figura **??**. Onde apenas se altera a data e o nome da pessoa/entidade que criou o teste.
2. Secção *Frontmatter*: esta secção contém descrição e os metadados associados ao teste presente. Para este efeito utiliza palavras-chave para organização dos dados. A secção é delimitada pela denotação de comentário `/*---` e `---*/`.
3. Secção *Body*: a última secção do teste vai conter o código a ser executado. O mesmo utiliza algumas funções auxiliares que facilitam o processo de testes.

A secção do *Frontmatter* emprega palavras chave para guardar a metadata do teste. Cada palavra chave é associada a uma string com a informação relativa do teste. De seguida apresentamos uma lista com todas as possíveis palavras chave. juntamente com o respectivo significado:

1. *description* - contém uma pequena descrição sobre o que está a ser testado;
2. *esid* - contém um identificador hash da porção do ECMAScript associado à feature a ser testada (o identificador deve ser referente à versão mais recente do standard no momento da criação do teste);
3. *info* - contém uma descrição mais profunda do comportamento esperado do teste, frequentemente incluí uma citação direta do standard;
4. *negative* - indica que o teste espera lançar um erro; a palavra chave têm associado um objecto com o tipo de erro que é suposto ser lançado (ex. `TypeError`, `ReferenceError`) e com a fase na qual o erro é lançado (ex. *parse* vs *resolution* vs *runtime*);
5. *includes* - contém uma lista com os ficheiros da harness que devem ser incluídos na execução dos testes;
6. *author* - contém a identificação do autor do teste;
7. *flags* - contém uma lista de booleanos para cada propriedade. Estas propriedades são: 1 *onlyStrict*, o teste apenas é executado em modo *strict*; 2 *noStrict*, o teste apenas é executado em modo "*sloppy*"; 3 *module*, o teste deve ser interpretado como um modulo *JavaScript*; 4 *raw*, executa o teste sem nenhuma modificação, o que implica que corre como *noStrict*; 5 *async*, apenas para testes assíncronos; 6 *generated*, flag que denota os ficheiros criados pelo teste; 7 *CanBlockIsFalse* e 8 *CanBlockIsTrue*, denotam o valor da propriedade `CanBlock` do agente;
8. *features* - contém uma lista com as features que são usadas no teste.
9. *es5id*/*es6id* - indicam que a feature do teste a ser testada pertence a versão ES5/ES6 do standard e apresenta o identificador hash da secção correspondente; estas duas palavras chaves foram descontinuadas e substituídas pela alavra chave *esid*.

O formato da metadata da *test262* está em constante mudança, acompanhando a evolução do standard ECMAScript. No entanto, estas mudanças não foram aplicadas retroativamente para todos os testes existentes. Deste modo, nem todos os testes estão anotados com todas as palavras chave apropriadas. Por exemplo: 1 existe um largo conjunto de testes anotados com as palavras chave *es5id* e *es6id*, apesar das mesmas já terem sido descontinuadas; 2 existe um largo conjunto de testes que não possuí a palavra chave *description*.

A Figura 3.1 mostra um teste da *test262* que testa o comportamento das funções asincronas. As linhas 1-2 contêm a secção de direitos de autor, as linhas 4-9 contêm a secção de *Frontmatter* e as linhas 11-12 contêm a secção de *body*. A secção de *Frontmatter* apenas contém as palavras chave: *author*, *esid*, e *description*. A metadata apresenta a falta de várias palavras chave, com é exemplo:

```
1  // Copyright 2016 Microsoft, Inc. All rights reserved.
2  // This code is governed by the BSD license found in the LICENSE file.
3
4  /*---
5  author: Brian Terlson <brian.terlson@microsoft.com>
6  esid: sec-async-function-objects
7  description: >
8    %AsyncFunction% exists and is a function
9  ---*/
10
11 var AsyncFunction = async function foo() {}.constructor;
12 assert.sameValue(typeof AsyncFunction, "function");
```

Figura 3.1: Function for an Async Function

async para indicar que o teste utiliza a feature assincrona da linguagem, e *includes* para incluir os modulos da harness que o teste necessita para executar o teste.

3.2 Cálculo da Metadata - Construções Sintáticas

O cálculo das construções sintáticas presentes no teste é feita através da ferramenta *esprima*, que permite-nos analisar a árvore sintática do teste. A árvore sintática de um código em javascript é representado por um JSON, onde a chave *type* contém o nome da construção sintática. Uma vez adquirida a árvore sintática do teste, a obtenção das construções sintaticas utilizadas no mesmo torna-se uma tarefa trivial.

No entanto, a árvore sintática do teste é extensa até para um teste pequeno, como é exemplificado na Figura 3.2, onde podemos observar um teste, contendo apenas uma linha de código, e a sua árvore sintática apresentada na Figura 3.3, que se extende por mais de cinquenta linhas. No exemplo apresentado, após a análise da árvore sintática obtemos as seguintes construções sintáticas:

- *Program*
- *ExpressionStatement*;
- *CallExpression*;
- *MemberExpression*;
- *Identifier*;
- *NewExpression*;
- *ArrowFunctionExpression*;

```
Object.seal(new (Object.getPrototypeOf(() => {}).constructor)());
```

Figura 3.2: Exemplo de um teste para a função seal do objecto Object

Figura 3.3: Árvore sintática do teste exemplo

- *BlockStatement*.

De forma a apanhar todas as construções sintáticas a árvore sintática é recursivamente analisada. Começando no topo da árvore e para todos os parâmetros da atual construção sintática o programa é novamente chamado. Caso a construção sintática atual ainda não tenha sido utilizada é acrescentada a um *array*. No final da análise, o *array* contém todas as construções sintáticas chamadas na árvore sintática do teste.

Os problemas ao executar este cálculo são as limitações do próprio *esprima*. Estas limitações são o suporte na totalidade apenas da versão 7 do standard ECMAScript e os testes que estam implementados de forma a retornar resultados negativos, os quais o *esprima* não consegue analisar.

3.3 Cálculo da Metadata - Versão

Descobrir a versão a que cada teste pertence é a tarefa mais difícil no cálculo da nova metadata para os testes. Pois a versão pode ser determinada pela presença de um elemento, construção sintática ou *built-in*, de uma determinada versão ou por uma mudança de comportamento das funções. Sendo que para o cálculo da mesma foram utilizadas três abordagens:

1. Abordagem Dinâmica
2. Abordagem Estática
3. Abordagem Mista

As abordagens dinâmica e estática complementam-se uma a outra criando uma abordagem mista. Esta abordagem mista consiste na execução da abordagem dinâmica seguida da abordagem estática de forma a melhorar os resultados obtidos.

3.3.1 Abordagem Dinâmica

O método desenvolvido para a abordagem dinâmica baseia-se num modelo em cascata. Este modelo corre os testes numa implementação de referência para uma determinada versão, assumindo que os testes que obtiveram o resultado esperado pertencem à versão usada. Os testes que não têm o resultado esperado são novamente corridos na mesma implementação para a versão seguinte.

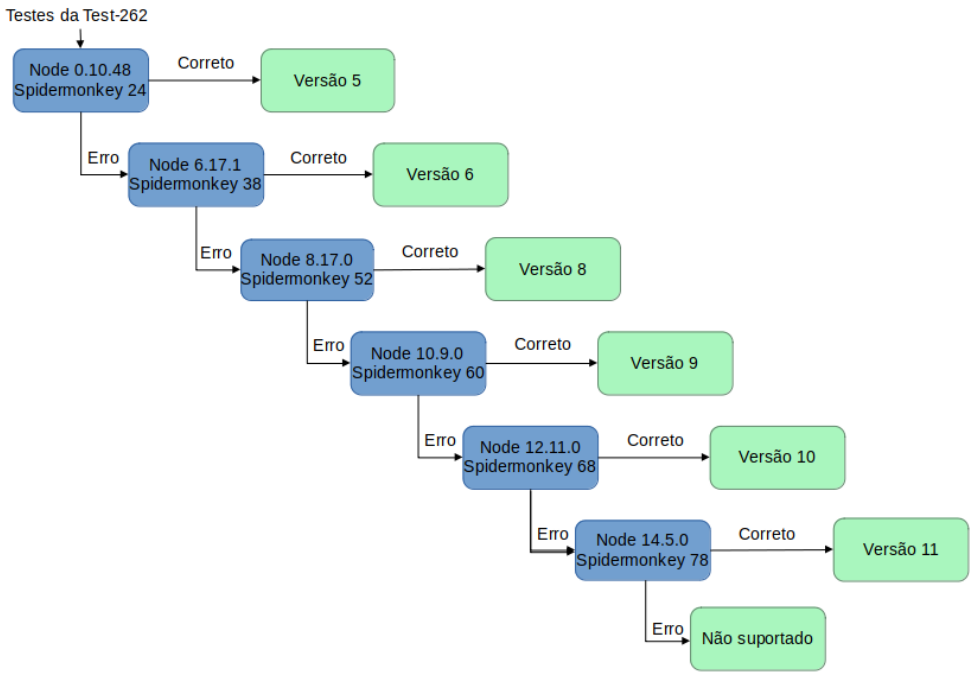

Figura 3.4: Diagrama sobre a abordagem dinâmica executada

No nosso trabalho são utilizadas seis versões de duas implementações de referência. As versões das implementações de referência correspondem às versões 5, 6, 8, 9, 10 e 11 do standard, a versão 7 do standard é decartada por ser pouco relevante e não ter introduzido novos *built-ins* nem novas funções associados aos *built-ins*. O método corre todos os testes da *Test262* para a versão da implementação

de referência correspondente à versão 5 do standard, os testes que passaram são associados à versão 5 do standard. Os testes que não tiveram a execução correta são novamente corridos para a versão da implementação de referência correspondente à versão 6. Voltando a ter o mesmo comportamento os testes que passam são associados à implementação correspondente à versão 6, os restantes testes passam para a implementação de referência da versão 8. O processo é repetido para as versões 9, 10 e 11. Para a versão 11 os testes que não passarem são considerados como não suportados por esta abordagem. O comportamento descrito é observado na Figura 3.4.

Na realização da abordagem dinâmica foram utilizados duas implementações de referência do javascript, o Node.js e o SpiderMonkey. Estas implementações foram escolhidos por implementarem o standard *ECMAScript* mais recente e terem mais versões lançadas, garantindo a existência implementação para cada versão do standard. No entanto estas versões podem estar incompletas, conter funcionalidades de uma versão mais recente do standard e/ou faltar funcionalidades referentes à versão.

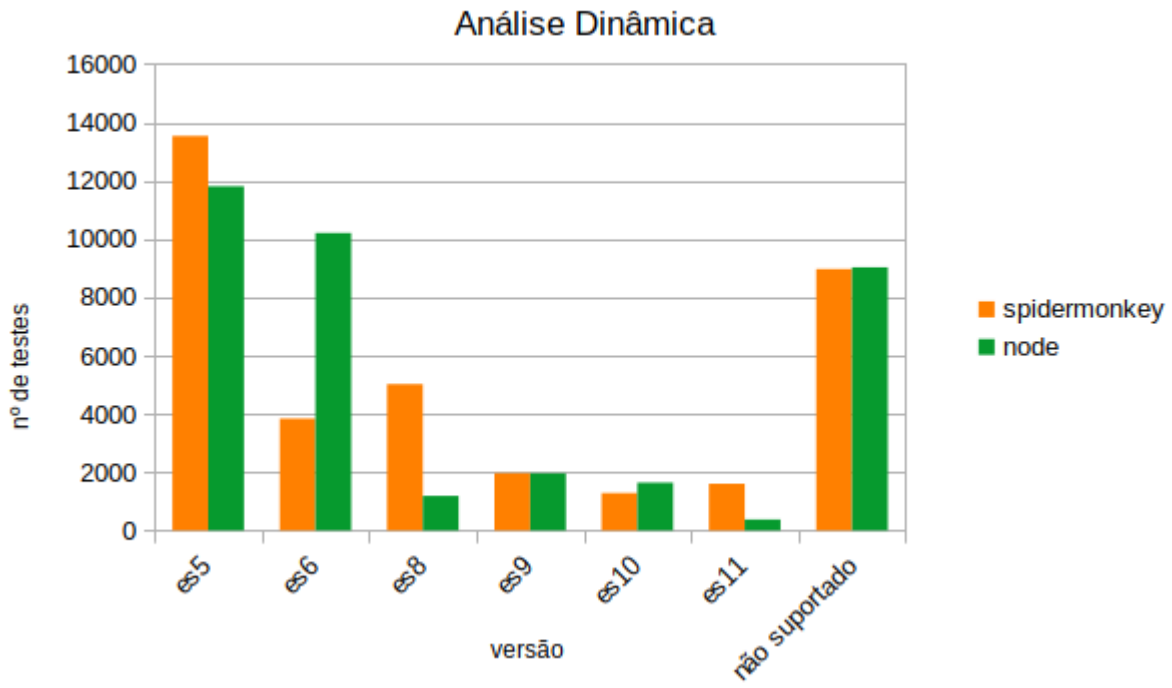

Figura 3.5: Comparação entre os resultados da análise dinâmica do SpiderMonkey e do Node.js

Através desde modelo foram obtidos os resultados presentes na figura 3.5. Onde podemos comparar os resultados das implementações do Node.js e do SpiderMonkey, a verde e a laranja respetivamente. Como pode ser concluído os resultados apresentam diferenças significativas nomeadamente para as versões 5, 6, 8 e 11. O SpiderMonkey filtra mais testes para as versões 5 e 8 do standard *ECMAScript* enquanto que o Node.js filtra um grande número de testes para a versão 6 do standard. Os resultados dinâmicos do Node.js aparentam estar mais corretos uma vez que a versão 6 do standard é

a versão que introduz um maior número de built-ins e funções que se traduz num grande aumento do número de páginas como é visto na Figura 1.1 o que faz aumentar o número de testes da bateria de testes.

3.3.2 Abordagem Estática

A metodologia da abordagem estática baseia-se na análise da árvore sintática do teste, recorrendo à ferramenta *esprima* para obter a mesma. A árvore sintática do teste permite-nos analisar todos os objetos, funções, propriedades, operatores, variáveis e construções sintáticas presentes no teste. A abordagem estática procura a versão mais antiga do standard que possuí todos os objetos, funções, propriedades, operatores, variáveis e construções sintáticas.

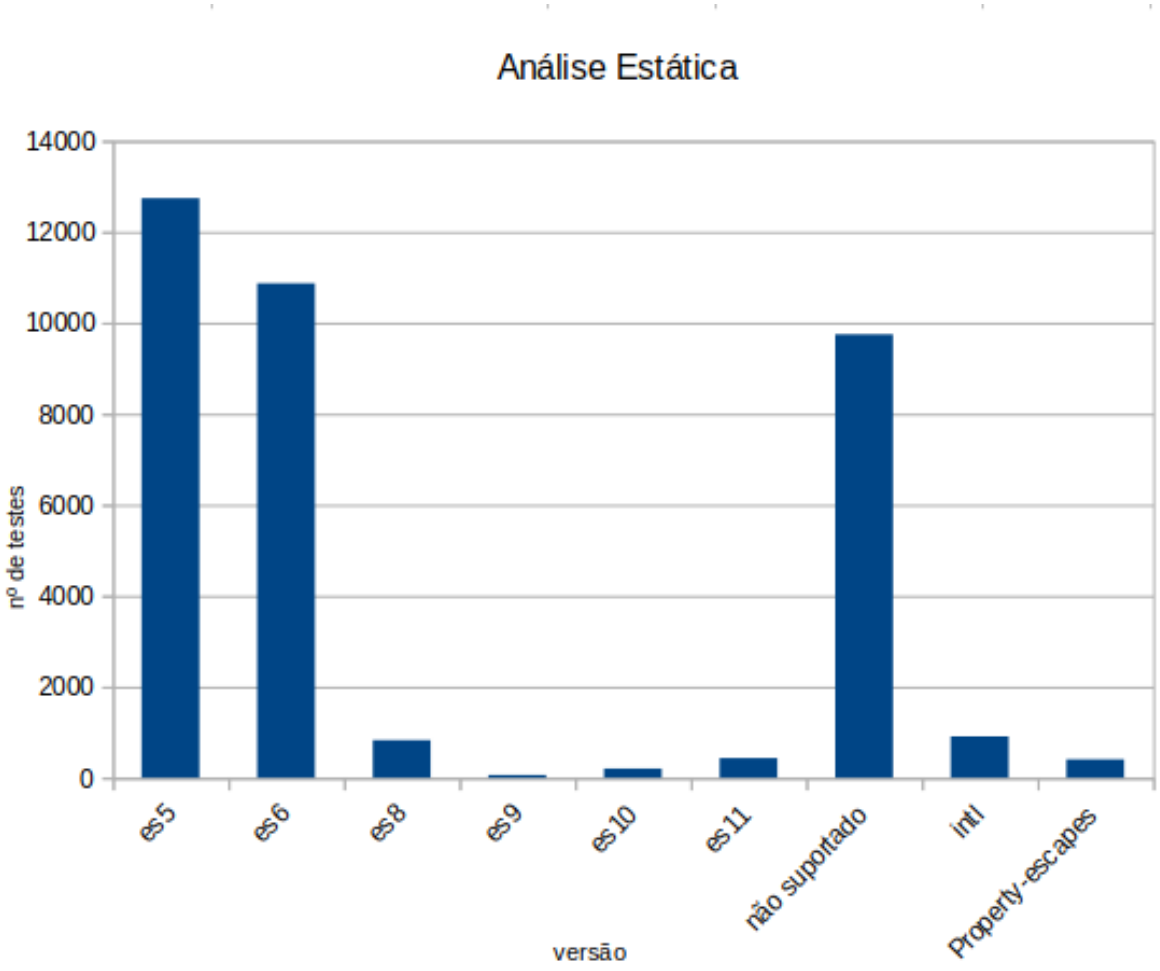

Figura 3.6: Resultados da análise estática do cálculo da versão

Para a abordagem estática foi criado um documento JSON com um array de objetos, em que cada elemento no array é um objeto correspondente a cada versão do standard. Cada versão tem associado

a si as variaveis globais, os operatores e as construções sintáticas presentes no standard da versão. Adicionalmente, também tem os built-ins, e as funções e propriedades dos mesmos.

Este ficheiro permite, ao analisar a árvore sintática do teste, encontrar os objetos, funções, variavéis, operatores e construções sintáticas presentes no teste associando a versão mais antiga que engloba todos os objetos, funções, variavéis, operatores e construções sintáticas. Ao descobrir a versão é necessário ainda garantir que no teste as funções inicializadas no mesmo não sejam tidas em consideração para o cálculo da versão.

Inicialmente foi corrida a analise estática e obtidos os resultados representados na Figura 3.6 que apresentam semelhanças à analise dinâmica para a implementação Node.js. Apresentando um grande número de testes para as duas primeiras versões do standard. Este grande número de testes deve-se a versão 5 ser a inicial e a versão 6 ter introduzido muitos objetos novos como é o caso do *Promise* e *Proxy*.

3.3.3 Abordagem Mista

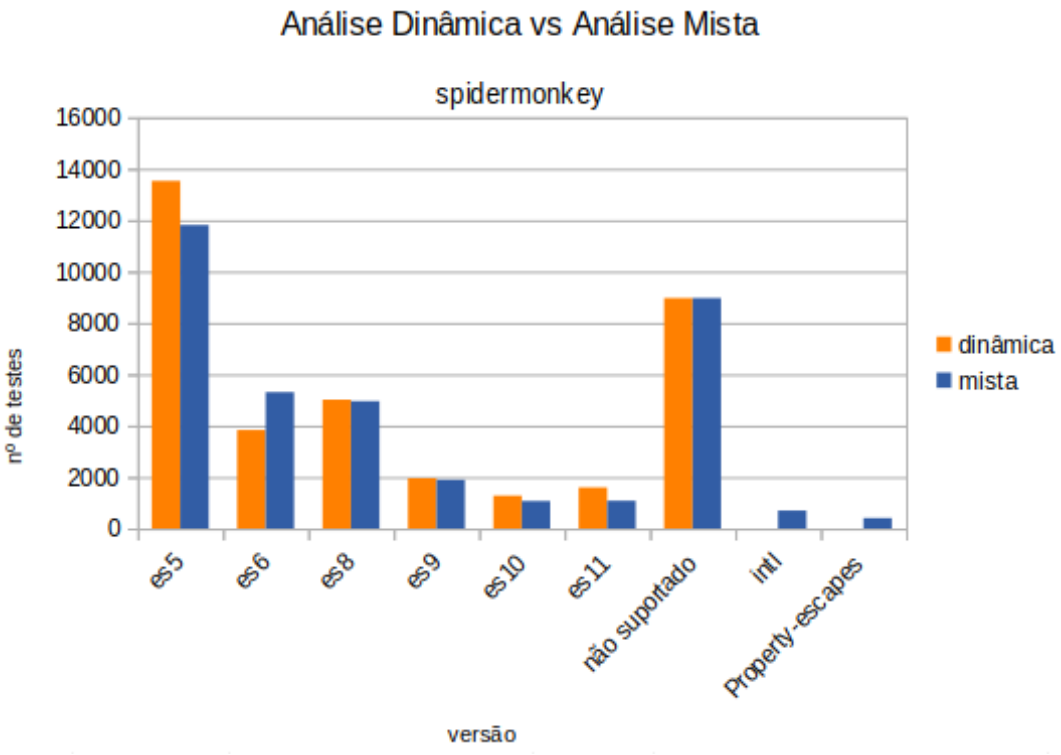

Figura 3.7: Comparação dos resultados da análise dinâmica versus análise mista para o SpiderMonkey

A abordagem mista é uma filtragem dupla onde as duas abordagens descritas acima são usadas uma após a outra de forma a melhorar os resultados de ambas as análises. Assim sendo foi corrida a abordagem dinâmica e aos resultados desta abordagem são novamente filtrada com a abordagem

estática, verificando que os testes não contem objetos, funções ou propriedades que apenas foram introduzidas em versões mais recentes do standard. Esta análise mista consiste em analisar os resultados da analise dinâmica para cada versão e confirmar que não existem parametros introduzidos numa versão posterior. Os resultados desta análise mista estão representadas nas Figura 3.7 e Figura 3.8 onde são comparados os resultados dinâmicos com os resultados mistos para as duas implementações, SpiderMonkey e Node.js respetivamente.

Como pode ser observado na Figura 3.7 os resultados da análise mista para o SpiderMonkey aproximam-se dos resultados esperados, havendo uma grande redução do número de testes filtrados para a versão 5 do standard. No entanto os resultados para a versão 6 do standard ainda aparentam ser escassos apesar de haver uma melhoria no número de teste filtrados.

Os resultados da análise mista para a implementação do Node.js, apresentado na Figura 3.8, demostra variações ligeiras em relação à análise dinâmica do Node.js. Havendo uma pequena redução do número de testes da versão 5 do standard que foram filtrados com a versão 6 do standard. Outras reduções devem-se aos testes pertencerem as propostas *intl* e *property-escapes*. As diferenças entre as análises dinâmicas e análises mistas reinforçam a ideia da implementação do Node.js ser mais correta do que a implementação do SpiderMonkey, pois a análise mista filtra menos testes. Contrariamente a análise mista beneficia mais a implementação do SpiderMonkey do que a implementação do Node.js.

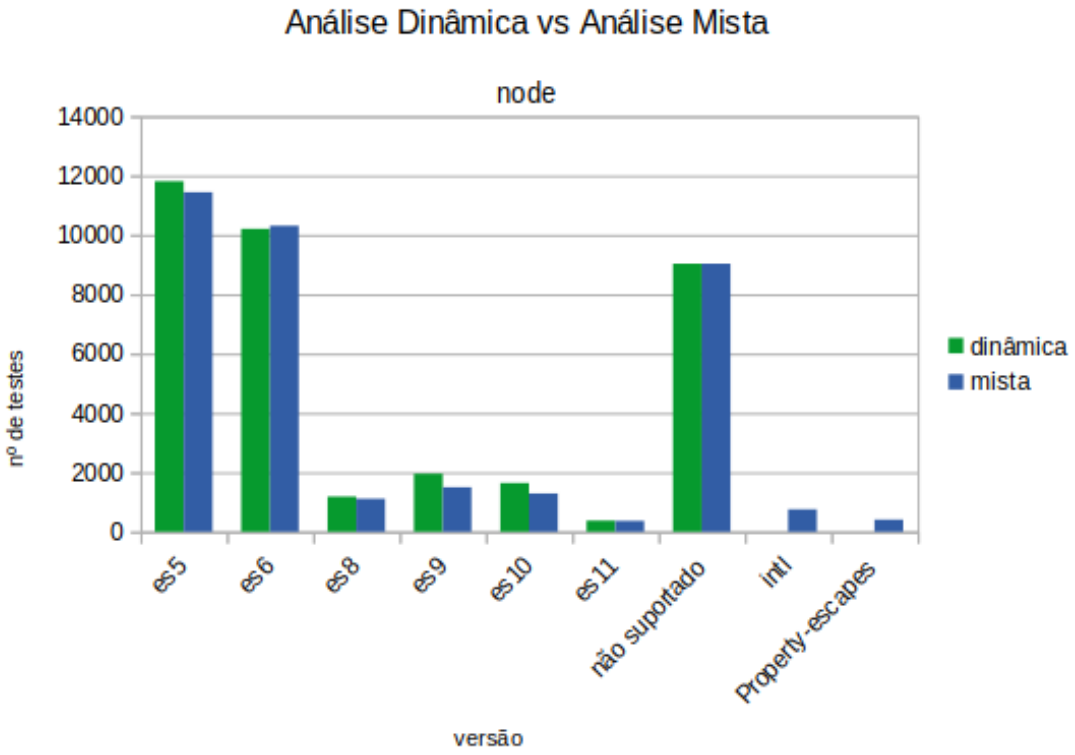

Figura 3.8: Comparação dos resultados da análise dinâmica versus análise mista para o Node.js

3.4 Cálculo da Metadata - Built-ins

De forma a calcular os built-ins utilizados em cada teste foram utilizados dois métodos distintos para obter os resultados. O primeiro dos quais utiliza uma análise baseada em palavras chaves com recurso a ferramenta *esprima*. O segundo método utiliza uma abordagem dinâmica para encontrar todas as funções associadas aos built-ins utilizados, bem como os próprios built-ins utilizados em cada teste.

3.4.1 Baseado em palavras-chave

Esta abordagem analisa a árvore sintática do teste para encontrar os built-ins utilizados e as funções e propriedades do built-in utilizadas. A análise dos built-ins utilizados no teste é feita ao encontrar as chamadas dos built-ins identificados pela construção sintática *Identifier* com o nome do mesmo. Adicionalmente é através da construção sintática *MemberExpression* que vamos conseguir retirar as funções e as propriedades referentes aos built-in utilizados confirmando se as propriedades/funções existem, no standard, para o built-in em questão. Sendo que a construção sintática *MemberExpression* se refere ao acesso de uma função ou propriedade de um objeto. É ainda necessário confirmar que as propriedades/funções encontradas são definidas no standard e não no próprio teste de forma a evitar obter propriedades/funções que não existam no standard.

Ao verificarmos a existência das construções sintáticas *ObjectPattern* ou *ObjectExpression* retiramos que o teste utiliza o built-in *Object*. Semelhantemente se existirem as construções sintáticas *ArrayPattern* ou *ArrayExpression* também concluimos que o teste utiliza o built-in *Array*.

Através desta análise para o teste da Figura 3.9 obtemos que foram utilizados os built-in *Array*, *Object* e *Function*, onde os dois primeiros são obtidos pelos *Identifiers* na linha 16 e 20 respetivamente. O ultimo built-in apresentado é utilizado ao executar a função *call* que é referente ao built-in *Function* (linha 20). As propriedades e funções que cada built-in utiliza neste teste são para o *Array* a propriedade *prototype*, para o *Object* a propriedade *prototype* e a função *hasOwnProperty*, e por fim para o *Function* a função *call*.

```
15  //CHECK#1
16  Array.prototype.myproperty = 42;
17  var x = Array();
18  assert.sameValue(x.myproperty, 42);
19
20  assert.sameValue(Object.prototype.hasOwnProperty.call(x, 'myproperty'), false);
21
```

Figura 3.9: Teste exemplo para o objecto Array

3.4.2 Abordagem Dinâmica

Na abordagem dinâmica foi feita uma refactorização das funções do standard de forma a que sempre que uma função é chamada a mesma acrescenta num *Array* de resultados o seu nome, caso ainda não esteja incluída, e só depois correr a função normalmente. No final da execução do testes o *Array* de resultados contem todas as funções chamadas no decorrer do teste.

Nesta refactorização foi utilizado um código para através do html do standard serem retiradas as funções dos objetos automáticamente. Uma vez conhecidas as funções foi corrido outro código que cria um ficheiro com todas estas funções refactorizadas pela forma representada na Figura 3.10, onde $1 representa o built-in e $2 representa o nome da função, e `log42` é o *Array* que contem todas as funções chamadas. A função `copyArgs` guarda os argumentos num *Array* para ser usado no método *apply__*.

```
$1.$2__ = $1.$2;
$1.$2 = function () {
    log42.indexof__(\"$1\.$2\") === -1 ? log42.push__(\"$1\.$2\") : null;
    var args = []
    args = copyArgs(arguments, args);
    return $1.$2__.apply(this, args)
```

Figura 3.10: Estrutura padrão do wrapper para a análise dinâmica do cálculo dos built-ins

Na figura 3.11, é observada a implementação do wrapper para a função `prototype.toString` do objeto `Object`. Após a obtenção dos wrappers para todas as funções, os testes foram executados na versão mais recente do Node.js com os wrapper, a função `copyArgs` e a inicialização do *Array* `log42` em prefixo no teste. Ao terminar a execução é retirado o *Array* `log42` com todas as funções chamadas na execução do teste.

```
Object.prototype.toString__= Object.prototype.toString;
Object.prototype.toString = function (){
    log42.indexOf__("Object.prototype.toString") === -1 ? log42.push__("Object.prototype.toString") : null;
    var args = []
    args = copyArgs(arguments, args);
    return Object.prototype.toString__.apply__(this, args)
}
```

Figura 3.11: Exemplo do wrapper para o teste exemplo

Os testes que terminam com erro, isto é os que possuem na metadata o campo *"negative"*, não são tidos em conta para este abordagem pois devido ao teste retornar erro não é possível obter o *Array*. Testes que verificam as propriedades das funções, como o nome da função ou o número de argumentos que recebe, também não teem a execução correta, por isso não são considerados nesta abordagem.

4

Caracterização da Bateria de Testes Test262

Conteúdo

No decorrer desta tese tivemos em grande foco a bateria de testes Test262 de modo que foi construida uma caracterização da mesma. Analisando os testes da bateria separando os mesmos por directórios, com maior atenção para os directórios mais importantes, *built-ins* e *language*. Os testes pertencentes ao directório *built-ins* foram separados por grupos de *built-ins*. Esta divisão permite uma analise mais detalhada dos testes para cada grupo de *built-ins*. Os grupos de *built-ins* são retirados diretamente das secções do standard e correspondem a:

- *Global* - contém os testes relativos ao objeto *Global*;
- *Fundamental* - contém teste das funções ensenciais da linguagem, os objetos *Object*, *Function*, *Boolean*, *Symbol* e *Error*;
- *Number and Dates* - contém os testes de todos os objeto relacionados com números, ou seja engloba os testes para os objetos *Number*, *BigInt*, *Math* e *Date*;
- *Text Processing* - contém os testes para os objetos referentes ao processamento de texto, os objetos *String* e *RegExp*;
- *Indexed Collection* - contém os teste para os objetos que guardam valores pelo valor do indice, os objetos *Array* e *TypedArray*;
- *Keyed Collection* - contém os testes dos objetos que indexem valores por chave, os objetos *Map*, *Set*, *WeakMap* e *WeakSet*;
- *Structured Data* - contém os testes das estruturas que armazenam dados fazem parte deste grupo, os objetos *ArrayBuffer*, *SharedArrayBuffer*, *DataView*, *Atomics* e *JSON*
- *Control Abstraction* - contém os teste que , os objetos *Iterator*, *AsyncIterator*, *GeneratorFunction*, *AsyncGeneratorFunction*, *Generator*, *AsyncGenerator*, *Promise* e *AsyncFunction*;
- *Reflection* - contém os testes para os objetos *Reflect* e *Proxy*;
- *Managing Memory* - contém os objetos que só foram introduzidos na versão 12 do standard que retratam a gestão de memória na linguagem javascript. Estes objetos são *WeakRef* e *FinalizationRegistry*
- *URI* - contém os testes referentes ao URI, os objetos *decodeURI*, *encodeURI*, *decodeURIComponent* e *encodeURIComponent*

Para o directório *language* foram apenas analisados os testes dos directórios *expressions* e *statements*, uma vez que estes são os mais importantes na implementação do standard. Os restantes testes são analisados em conjunto designados pelo campo *other*.

Tabela 4.1: Número de Testes por Grupo

		Nº de Testes
Built-ins	**Global**	212
	Fundamental	4146
	Numbers and Dates	1473
	Text Processing	2647
	Indexed Collections	4620
	Keyed Collection	525
	Structured Data	1133
	Control Abstraction	894
	Reflection	463
	Managing Memory	89
	URI	172
Language	**Statements**	9039
	Expressions	10511
	Other	3009

Na Tabela 4.1, estam representadas as divisões dos testes, bem como o número de testes que cada divisão contem. Ao analisar a tabela concluimos que os grupos do directório *built-in* com maior número de testes são os grupos *Fundamental*, *Indexed Collections* e *Text Processing* todos eles com mais de 2500 testes o que se deve aos objetos destes grupos terem sido introduzidos nas primeiras versões do standard ECMAScript. No caso dos grupos referentes ao directório *language* pode-se observar que os grupos *expressions* e *statements* são de factos os mais importantes visto que cada grupo tem mais do triplo dos testes de todas as outras pastas juntas.

4.1 Número de Linhas

O número de linhas dos testes foi analisado em maior detalhe. Para esta análise foram obtidos o primeiro quartil, a mediana e o terceiro quartil do número de linhas de cada grupo que formam o gráfico representado na Figura 4.1. Para os bigodes da Figura 4.1 é usado o valor mínimo e máximo que não ultrapassem o 1º quartil menos o IQR e o 3º quartil mais o IQR, respectivamente. O IQR representa o alcance interquartil, que é calculado pela diferença entre o terceiro e o primeiro quartil vezes 1.5. Adicionalmente foi também obtida a média, o número máximo, o número minímo e o desvio padrão para o número de linhas dos testes, estes valores são apresentados na Tabela 4.2.

Os grupos *global* e *fundamental* apresentam valores bastante semelhantes tanto para os valores da Tabela 4.2 e dos valores da Figura 4.1, sendo a média de cerca de onze linhas por teste, o teste com mais linhas é para o *global* 107 e para o *fundamental* 116. No entanto o grupo *global* apresenta um desvio padrão de quase o dobro do desvio padrão do grupo *fundamental*. No gráfico da Figura 4.1 os valores dos dois primeiros grupos também são semelhantes onde os valores do grupo *global* são ligeiramente inferiores ao do grupo *fundamental*, ambos com uma diferença pequena entre o terceiro e

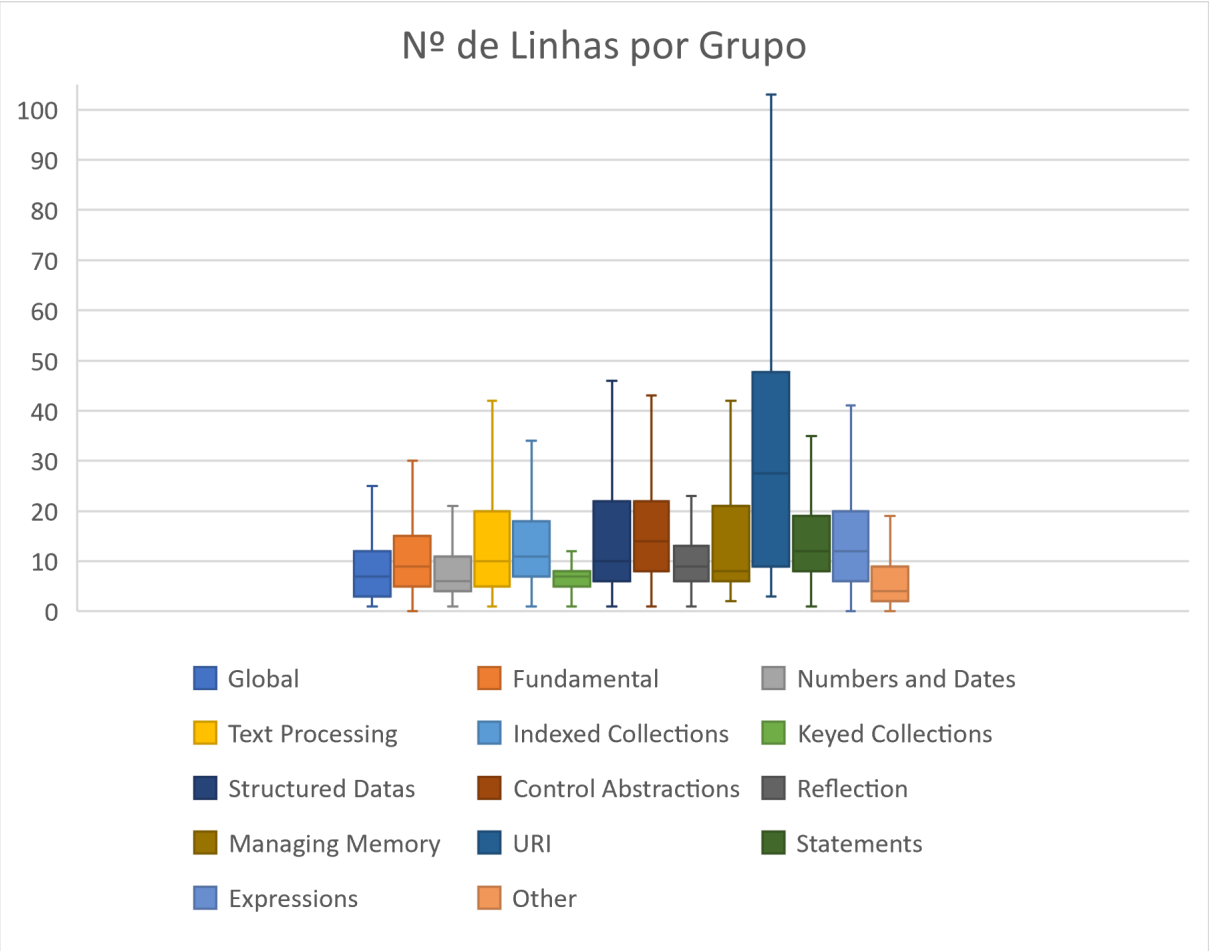

Figura 4.1: Diagrama de caixa para o número de linhas

o primeiro quartil onde estam presentes metade dos testes do grupo.

O grupo *Numbers and Dates* apresenta média de número de linhas dos testes na casa das 12 linhas e com um desvio padrão do número de linhas de quase 21. O teste deste grupo com número de linhas mais alto é de 210 linhas. Os quartis deste grupo estão bastante próximos uns dos outros indicando que metade dos testes neste grupo encontram-se entre um intervalo de linhas pequeno entre as 4 e 11 linhas de código.

Dois grupos que apresentam os mesmos valores de mediana são os grupos *Text Processing* e *Structured Data*, tendo também os valores do primeiro e do terceiro quartil bastante semelhantes. O que demonstra que a maioria dos testes se encontra neste intervalo. No entanto, o grupo *Text Processing* tem vários testes com número de linhas bastante mais elevado o que faz a média e desvio padrão muito superiores aos valores do grupo *Structured Data*, mais especificamente a média quase duas vezes maior e o desvio padrão quase oito vezes superior. O teste com maior número de linhas do grupo *Text Processing* é de 1703 linhas, sendo o grupo com o segundo maior número de linhas.

Tabela 4.2: Dados Sobre o Número de Linhas

		Média	Desvio Padrão	1º Quartil	Mediana	3º Quartil	Mín.	Máx.
Built-ins	**Global**	11.73	15.96	3	7	12	1	107
	Fundamental	10.87	7.93	5	9	15	0	116
	Numbers and Dates	12.07	20.57	4	6	11	1	210
	Text Processing	32.66	126.49	5	10	20	1	1703
	Indexed Collections	14.06	33.21	7	11	18	1	2061
	Keyed Collection	8.15	5.65	5	7	8	1	46
	Structured Data	16.66	16.40	6	10	22	1	165
	Control Abstraction	17.00	13.22	8	14	22	1	87
	Reflection	10.73	7.15	6	9	13	1	52
	Managing Memory	12.51	10.08	6	8	21	2	48
	URI	29.05	21.35	9	27.5	47.25	3	103
Lang.	**Statements**	16.78	17.55	8	12	19	1	179
	Expressions	18.71	58.69	6	12	20	0	1585
	Other	7.33	9.88	2	4	9	0	144

Indexed Collections contém o teste com maior número de linhas, com 2061 linhas. No entanto, a maioria dos testes encontra-se com um baixo número de linhas, tendo a mediana de 11 linhas, o primeiro quartil de 7 linhas, o terceiro quartil de 18 linhas e a média das linhas dos testes de 14 linhas. Isto demonstra que a maioria dos testes desde grupo não tem muitas linhas mas existe um grande número de testes com número de linhas bastante elevado, o que também é visível no desvio padrão alto de 33 linhas.

O grupo *Keyed Collection* é o grupo com menor variância, o que se pode dever a ser o grupo com menor número de testes e onde nenhum dos testes apresenta um número elevado de linhas. Este grupo apresenta o desvio padrão mais curto de entre todos os grupos analisados, bem como o menor intervalo entre o primeiro e o terceiro quartil, e o menor número máximo de linhas.

Os testes do grupo *Control Abstration* têm um intervalo de linhas desde 1 linha até as 87 linhas, com o primeiro quartil nas 8 linhas e com o terceiro quartil nas 22 linhas, onde cerca de metade dos testes deste grupo se inserem. A média do número de linhas é de 17 linhas devido a haver muitos testes com número de linhas entre as 22 e as 87 linhas.

O grupo *Reflection* apresenta uma variância no número de linhas bastante baixo, a diferença entre o primeiro e o terceiro quartil é de 7 linhas, uma das diferenças mais pequenas de qualquer grupo. Este grupo é também um dos grupos com número máximo de linhas mais baixo, com 52 linhas de máximo. Constituindo o segundo grupo mais condensado em termos de número de linhas, isto é com menor diferença entre o primeiro e o terceiro quartil.

O grupo *Managing Memory* apresenta uma mediana de 8 linhas e um primeiro quartil bastante perto da mediana com 6 linhas, indiciando que quase um quarto dos testes neste grupo têm entre 6 a 8 linhas. No entanto, o terceiro quartil é de 21 linhas mostrando uma maior diversidade no número de linhas para metade dos testes. O número máximo de linhas de 48, sendo o segundo grupo com menor número máximo de linhas.

O grupo *URI* apresenta uma maior intervalo entre o primeiro e o terceiro quartil, com 9 e 47 linhas respectivamente. Este grupo apresenta também a maior mediana e a segunda maior média de todos os grupos analisados.

O grupo *statements* apesar de ser um dos grupos com mais testes, o intervalo entre o primeiro e o terceiro quartil continua a ser pequeno, entre 8 e 19, onde se encontram metade dos testes. Apresentando uma mediana de 12 linhas, uma média de quase 17 linhas e um desvio padrão de quase 18 linhas.

O grupo *language* contém a mesma mediana que o primeiro grupo de 12 linhas, mas com o intervalo entre o primeiro e o terceiro quartil ligeiramente maior que o primeiro grupo. Este grupo possui ainda testes com número de linhas bastante altos atingindo um máximo de 1585 linhas, contribuindo para um desvio padrão elevado embora a média se mantenha semelhante à do primeiro grupo com 19 linhas de média.

Por fim o grupo *other*, é o grupo com mediana e média mais baixos de todos os grupos, possuindo desvio padrão e quartis também bastante baixos. No entanto, este grupo contém alguns testes com muitas linhas onde o teste com maior número de linhas têm 144 linhas.

Concluindo, a grande maioria dos grupos analisados apresentam média de linhas na casa das 10-20 linhas de código, havendo 2 grupos com média acima (*Text Processing* e *URI*) e 2 grupos com média abaixo(*Keyed Collection* e *other*). No entanto, todos os grupos apresentam alguns testes com número de linhas bastante elevado destacando-se os grupos *Text Processing*, *Indexed Collection* e *expressions* que têm testes com mais de 1000 linhas de código, onde o grupo *Indexed Collection* têm máximo de 2061 linhas.

4.2 Número de *Asserts*

Na análise do número de *asserts* presentes nos testes de cada grupo retiramos os dados expostos na Tabela 4.3, onde são apresentadas a média, o desvio padrão, o valor mínimo e o valor máximo dos asserts presentes no grupo. Para uma melhor compreensão do número de *asserts* por grupo foram ainda retirados a mediana, o primeiro quartil e o terceiro quartil, de forma a criar o gráfico de diagrama de caixa representado na Figura 4.2 onde o bigode vai ser mais uma vez o IQR.

Os grupos *global*, *Numbers and Dates* e *Text Processing* apresentam o valor do primeiro quartil com zero indicando que um grande número de testes nos grupos. A mediana dos três grupos também é semelhante, onde os dois primeiros têm mediana de 1 *assert* e o terceiro grupo têm mediana de 0 *asserts*, sendo um dos grupos com maior percentagem de testes com 0 *asserts*. No entanto, o terceiro grupo apresenta um número máximo de *asserts* muito alto possuindo um teste com 103 *asserts* sendo o segundo grupo no total com um maior número máximo. O facto de ter testes com muitos *asserts* vai

Tabela 4.3: Dados Sobre o Número de Asserts

		Média	Desvio Padrão	1º Quartil	Mediana	3º Quartil	Mín.	Máx.
Built-ins	**Global**	2.52	4.20	0	1	3	0	20
	Fundamental	1.53	1.71	1	1	2	0	31
	Numbers and Dates	2.15	4.53	0	1	2	0	72
	Text Processing	1.54	5.29	0	0	1	0	103
	Indexed Collections	2.24	2.66	1	1	2	0	32
	Keyed Collection	2.19	1.97	1	2	2	0	16
	Structured Data	3.62	4.49	1	2	4	0	30
	Control Abstraction	2.42	2.95	1	1	3	0	18
	Reflection	2.06	2.04	1	1	2	0	20
	Managing Memory	3.34	3.51	1	2	6	0	13
	URI	0.07	0.33	0	0	0	0	2
Lang.	**Statements**	3.20	4.00	1	2	4	0	55
	Expressions	3.24	7.41	0	2	4	0	306
	Other	1.24	2.83	0	0	2	0	35

elevar o valor do desvio padrão do grupo sendo também um dos mais altos dos grupos analisados.

O grupo *fundamental* apresenta um primeiro quartil de 1 *assert*, igual à sua mediana o que indica que mais de um quarto dos testes do grupo têm 1 *assert*. O terceiro quartil do grupo é de 2 *asserts*, Tanto a média como o desvio padrão do grupo apresentam valores a rondar 1,6 com média nos 1,5 e o desvio padrão de 1,7. O valor máximo de *asserts* do grupo é de apenas 31 *asserts*.

O grupo *global* apesar de apresentar número máximo de *asserts* mais baixo dos quatro grupos acima mencionados é o que apresenta média mais elevada, com média de 2,5 *asserts*. Enquanto que os grupos *fundamental* e *Text Processing* têm média de 1,5 *asserts*, e o grupo *Numbers and Dates* têm média de 2 *asserts*.

No grupo *Indexed Collections* a mediana e o primeiro quartil possuem o mesmo valor de 1 *assert*, de onde concluímos que mais de um quarto dos testes do grupos possuem um único *assert*. Pela análise da média do grupo por ser maior que 2 *asserts* e do terceiro quartil ser perto da mediana, com 2 *asserts*, concluimos que há vários testes com número de *asserts* alto, não ultrapassando os 32 *asserts* por ser o valor máximo do grupo.

O grupo *Keyed Collections* apresenta valores da mediana e do terceiro quartil idênticos, que como no grupo anterior concluímos que mais de um quarto dos testes têm 2 *asserts* e que mais de metade dos testes do grupo têm 2 ou menos *asserts*. Este grupo apresenta um número máximo de *asserts* bastante baixo, com máximo de 16 *asserts*, que faz este grupo ser o terceiro com valor máximo mais baixo.

O grupo *Structured Data* é um dos grupos que apresenta maior diferença entre o primeiro e o terceiro quartil, 1 e 4 *asserts* respetivamente, e mediana de 2 *asserts*. Este grupo apresenta a média e o desvio padrão mais altos de todos os grupos (média de 3,6 e desvio padrão de 4,5) apesar do valor máximo de *asserts* no grupo é apenas de 30. Isto indica que um grande número de testes que apresentam valor mais alto que o terceiro quartil estão mais perto do valor máximo do que do terceiro quartil.

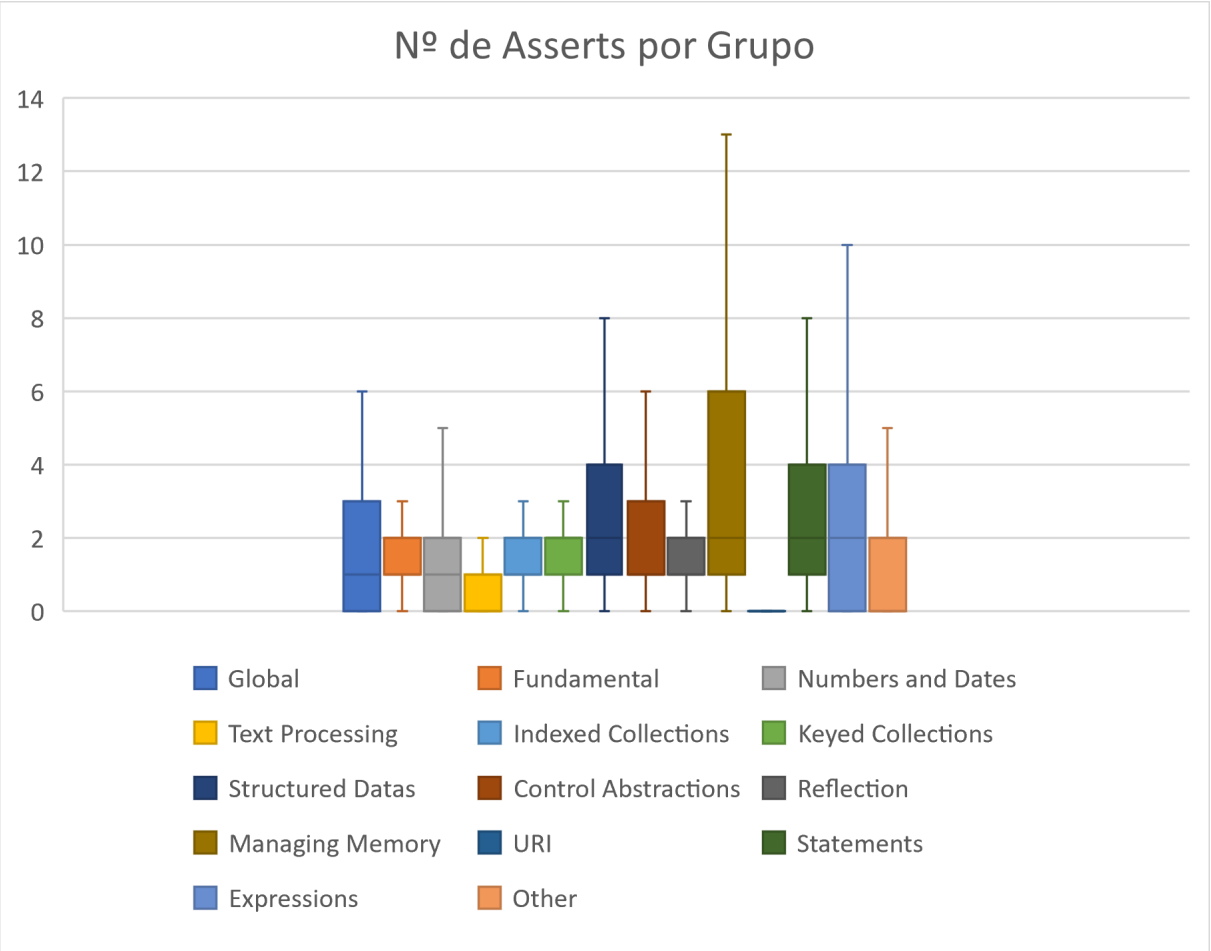

Figura 4.2: Diagrama de caixa para o número de asserts

Os grupos *Control Abstraction* e *Reflection* apresentam mediana igual ao primeiro quartil, com o valor de 1 *assert* por teste. A mediana ser igual ao primeiro quartil indica que pelo menos um quarto dos testes dos grupos têm o valor da mediana. As médias e desvio padrão de ambos os grupos são semelhantes ambos na casa dos 2 valores embora maior no grupo *Control Abstraction*. O valor máximo dos grupos também é bastante próximo sendo de 18 para o *Control Abstraction* e de 20 para o *Reflection*. No entanto, o terceiro quartil do grupo *Control Abstraction* é de 3 *asserts* e do grupo *Reflection* é de 2 *asserts*.

O *Managing Memory* é o grupo com maior diferença entre o primeiro e o terceiro quartil onde se encontram pelo menos metade dos testes do grupo. O primeiro quartil tem valor de 1 *assert*, enquanto que o terceiro quartil tem 6. A mediana deste grupo é de 2 valores, o valor mais alto de todas as medianas em conjunto com outros 4 grupos. Apesar deste grupo ter grande diferença entre os quartis é o segundo grupo com menor máximo com 13 *asserts*. Tanto a média como o desvio padrão do grupo se encontram na casa dos 3 *asserts*.

O grupo *URI* representa um caso particular onde a grande maioria dos testes não contêm nenhum *assert*. O que é representado pelo primeiro quartil, mediana e terceiro quartil terem todos 0 como valor. Devido a isso a média do grupo é de aproximadamente 0,07 e o desvio padrão de 0,33. Para além disso o teste com maior número de *asserts* é de apenas 2.

Os grupos *statements* e *expressions* do directório *language* têm a mesma mediana com valor de 2 *asserts* e terceiro quartil de 4 *asserts*. Contudo, o primeiro quartil é distinto apresentando 1 e 0 *asserts* respectivamente, logo o grupo *expressions* possuí mais testes com 0 *asserts*. O desvio padrão do grupo *expressions* é quase o dobro do desvio padrão do grupo *statements* com 4 para o *statements* e de 7,4 para o *expressions*. Esta diferença no desvio padrão deve-se ao grupo *expressions* ter testes com grande número de *asserts*. Onde o teste com maior número de *asserts* apresenta 306 *asserts*, sendo também o teste com mais *asserts* da bateria de testes. Enquanto que no grupo *statements* o teste com maior número de *asserts* é de apenas 55.

Concluindo, a grande maioria dos grupos têm em média um baixo número de *asserts*. Havendo apenas 4 grupos com média de *asserts* superior a 3 *asserts*, *Structured Data*, *Managing Memory*, *statements* e *expressions*. Estes grupos com média superior a 3, têm a maioria dos testes com menos de 2 *asserts*, pois a sua mediana é de 2 *asserts*. No entanto, os grupos possuem muitos testes com grande número de *asserts*, sendo os quatro grupos com maior terceiro quartil em todos superior a 4 *asserts*. O grupo *Managing Memory* destaca-se por ter um valor máximo de *asserts* bastante baixo mas muitos testes com mais de 6 *asserts*.

4.3 Número de *Errors*

Na análise do número de erros presentes nos testes dos grupos foi novamente construída uma tabela com a média, o desvio padrão, o valor mínimo e o valor máximo de *errors* presentes no grupo, representada na Tabela 4.4. O gráfico com os quartis e medianas apresentado nas outras secções não é apresentado para esta secção devido ao quartis serem na maioria dos grupos de zero, para o primeiro e para o terceiro quartil. O que se deve aos *Errors* serem um método secundário de verificação do comportamento do teste, sendo os *Asserts* o método preferencial para a verificação do comportamento nos testes.

O grupo *global* apresenta média de aproximadamente 1,25 *errors* por teste e um desvio padrão de quase 3 valores. O teste do grupo com maior número de *errors* contem 16 *errors*. O seguinte grupo, *fundamental*, apesar de ter um maior máximo com 33 valores apresenta uma média e um desvio padrão mais reduzidos que o primeiro grupo com 0,33 de média e 1,1 de desvio padrão.

O grupo *Numbers and Dates* é o grupo pertencente aos *built-ins* que contem o teste com maior número máximo de *errors*, com cerca de 48 valores. O grupo apresenta ainda o maior desvio padrão

Tabela 4.4: Dados Sobre o Número de Errors

		Média	Desvio Padrão	1º Quartil	Mediana	3º Quartil	Mín.	Máx.
Built-ins	**Global**	1.26	2.86	0	0	1	0	16
	Fundamental	0.32	1.10	0	0	0	0	33
	Numbers and Dates	1.51	4.73	0	0	2	0	60
	Text Processing	0.93	1.63	0	0	1	0	16
	Indexed Collections	0.32	1.61	0	0	0	0	21
	Keyed Collection	0.00	0.00	0	0	0	0	0
	Structured Data	0.01	0.09	0	0	0	0	2
	Control Abstraction	0.11	0.42	0	0	0	0	4
	Reflection	0.00	0.00	0	0	0	0	0
	Managing Memory	0.00	0.00	0	0	0	0	0
	URI	3.57	2.55	1	4	5	0	12
Lang.	**Statements**	0.25	1.76	0	0	0	0	30
	Expressions	1.15	17.51	0	0	0	0	528
	Other	0.43	1.98	0	0	0	0	33

de todos os grupos do directório *built-ins* com 4,74 de desvio padrão e uma média de 1,5 *errors* por teste.

Text Processing é retratado por ter média de quase 1 *error* por teste, sendo de 0,93. O desvio padrão do grupo é de 1,63 valores e um teste com número máximo de *errors* de 16 valores. O seguinte grupo é o *Indexed Collections*, este grupo têm média baixa com aproximadamente 0,31 de *errors* e desvio padrão de quase 1,6 valores. O máximo de *errors* num teste deste grupo é de 21 *errors*.

Os grupos *Keyed Collection*, *Reflection* e *Managing Memory* são os três grupos analisados em que nenhum dos testes contém nenhum *errors*. Por isso, a média de *errors*, o seu desvio padrão, bem como os valores mínimos e máximos são de zero valores.

O grupo *Structured Data* é o que apresenta menor média e desvio padrão que tem pelo menos um testes com *errors*, sendo a média de 0,01 e o desvio padrão de 0,09 valores aproximados. O grupo é também o grupo que apresenta valor máximo de *errors* mais baixo com apenas 2, excluindo novamente os grupos que nenhum teste contem *errors*.

O grupo *Control Abstraction* também apresenta valores médios e desvio padrão bastante baixos, com aproximadamente 0,11 e 0,42 de média e desvio padrão respectivamente. Estes valores indicam que a grande maioria dos testes deste grupo não têm *errors* e os poucos grupos que contém *errors* têm entre 1 e 4 *errors*, sendo que o valor máximo é de 4 *errors*.

O grupo *URI* é também o grupo que apresenta maior média de *errors*, com 3,57 *errors* de média. O desvio padrão do grupo é semelhantemente um dos maiores analisados com 2,56 valores e o número máximo de *errors* é de apenas 12.

O grupo *statements* apresenta média e desvio padrão de *errors* baixos com valores de 0,25 para a média e de 1,76 de desvio padrão. O máximo de *errors* queum teste contém neste grupo é de 30 *errors*. Uma vez que o valor máximo do grupo é alto e, a média e o desvio padrão do grupo serem baixos indica que os testes do grupo apresentam na grande maioria 0 *errors* mas que existem grupos

com um grande número de *errors*.

O grupo *expressions* tem a peculiaridade de conter o teste com maior número de *errors* da bateria de testes, e com um valor de 528 *errors* que é um valor 11 vezes superior ao maior máximo dos restantes grupos. O facto do máximo ser tão alto influencia o desvio padrão do grupo elevando o mesmo para 17,51 valores. Apesar do grande valor máximo do grupo, a sua média é de apenas 1,15 valores devido ao elevado número de testes com 0 *errors*.

Por fim o grupo *other* é semelhante ao primeiro grupo do directório *language* o grupo *statement*, com valores ligeiramente superiores. A média deste último grupo é de 0,43 valores enquanto que o grupo *statements* é de 0,25 valores. Por sua vez, o desvio padrão do grupo *other* é de 1,97 cerca de 0,2 valores superior ao do grupo *statements*. O valor máximo do grupo *other* é novamente superior ao do grupo *statements* por 3 *errors*, com 33 *errors* para o grupo *other* e 30 *errors* para o grupo *statements*.

Concluindo, o número de *errors* por teste nos grupos é em média bastante baixo existindo apenas quatro grupos com média superior a 1 *error* por teste. Os grupos *global*, *Numbers and Dates* e *expresssions* apresentam média ligeiramente superior a 1 que se deve a terem alguns teste com elevado número de *errors*, No entanto, estes três grupos têm mais de metade dos testes com 0 *errors*, tendo a mediana de 0. O grupo *URI* é interessante sendo o único grupo que apresenta maior média de *errors* do que de *asserts*.

4.4 Versões dos testes

Após o cálculo da versão pelos métodos descritos no capítulo anterior foram retiradas as versões dos testes. Através da analise das versões calculadas para os grupos já definidos anteriormente podemos observar a diversidade de versões nos grupos. Para cada grupo é possível observar os testes que que correspondem a cada versão do standard. O aumento do número de testes dos grupos é normalmente representado pela introdução de novos objetos e/ou funções no grupo, havendo na maioria dos casos uma pequena introdução de testes para as versões mais recentes que se devem à descoberta de comportamentos não testados nas versões anteriores. O grupo *Managing Memory* composto pelos objetos *WeakRef* e *FinalizationRegistry*, ambos introduzidos na versão 12 do standard, não é representado nesta análise. O que se deve à análise desenvolvida não contemplar a versão 12 do standard na qual são introduzidos os objetos do grupo *Managing Memory*. As percentagens de testes para cada versão de cada grupo estão representadas no gráfico da Figura 4.3.

Para o grupo *global* a grande maioria dos testes do grupo, quase 79%, pertencem à versão 5 do standard. Do resto dos testes 15% estão associados à versão 6, estes testes devem-se à introdução de muitos novos objetos e funções. Os testes filtrados para a versão 8 do standard representam apenas 1% dos testes do grupo. As versões 9, 10 e 11 não obtiveram nenhum teste neste grupo, devido ao

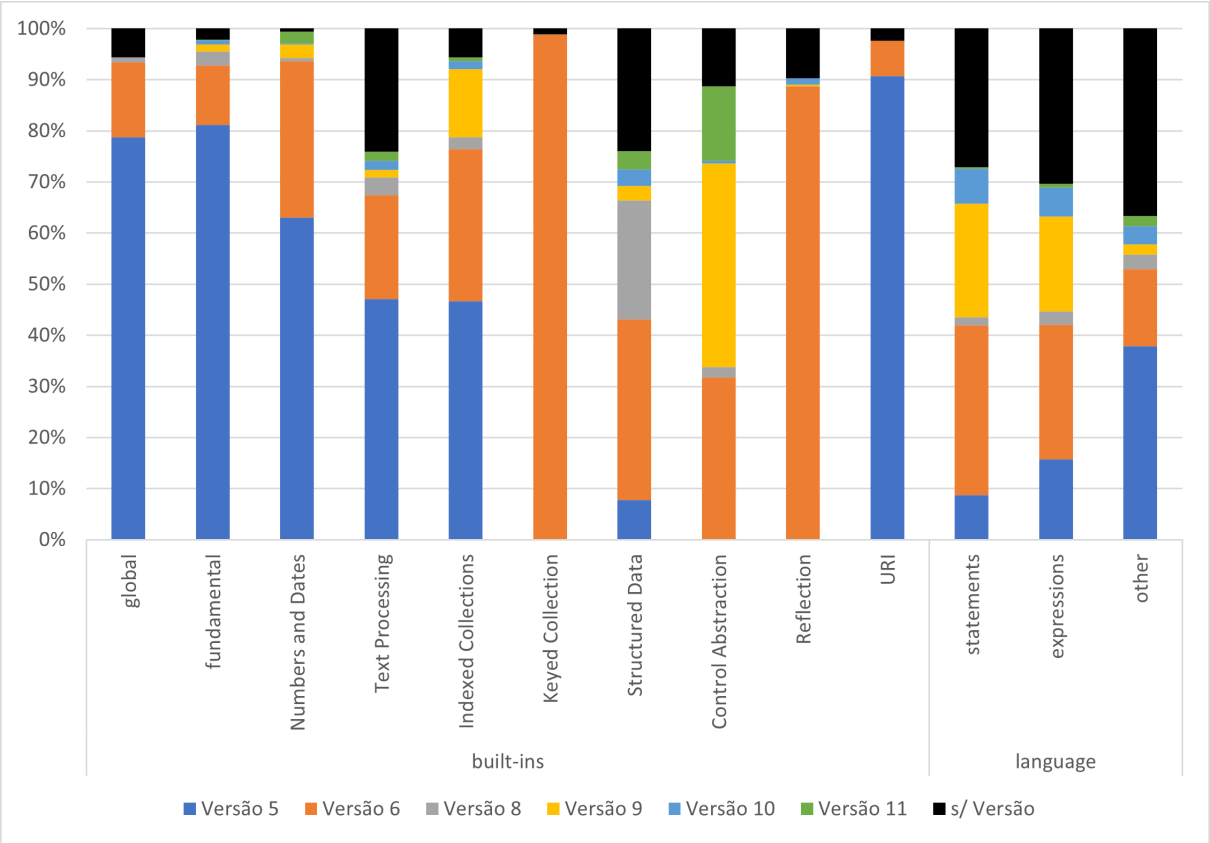

Figura 4.3: Diagrama para os resultados da versão

grupo conter o objeto *global* que é na sua maioria definido nas primeiras versões do standard. Para os restantes testes não foi obtida nenhuma versão no programa acima.

O grupo *fundamental* que é composto pelos objetos essenciais da linguagem é observado que a maioria dos testes pertence à versão 5 do standard. Mais especificamente 81% dos testes do grupo pertencem à versão 5 do standard, sendo que 17% dos testes estão divididos pelas outras versões com maior percentagem a ser filtrada para a versão 6 (11%). Contendo apenas 2% dos testes em que não foi possível encontrar nenhuma versão.

O grupo *Numbers and Dates* continua a apresentar a maioria dos testes com a versão 5 do standard, no entanto os testes da versão 5 têm menor peso no total do grupo. Apresentando 929 testes no total de 1473 testes do grupo, o que vai representar 63% do total de testes do grupo. A segunda maior fatia de testes do grupo está associada à versão 6 do standard com 30% dos testes, que se deve à introdução de várias funções para os objetos *Number* e *Math*, 6 e 17 funções respectivamente. A versão 11 do standard também introduz um novo objeto para o grupo com a introdução do objeto *BigInt*. Esta adição do objeto *BigInt* representa outra fatia, embora pequena, do número de testes. As versões 8 e 10 do standard não apresentam grande número de testes para o grupo, uma vez que não foi introduzido

nenhuma função, campo ou objeto para este grupo. tavendo também um pequeno número de testes em que não foi possível encontrar uma versão.

O grupo *Text Processing* semelhantemente aos grupos já descritos têm a sua maioria dos testes associados à versão 5 do standard. O grupo conta com 1247 testes para a versão 5 do standard, o que representa 47% do total dos testes do grupo. O grande número de testes para esta versão deve-se a terem sido introduzidos os objetos que formam o grupo *Text Processing* nesta versão. O grupo é composto ainda por 537 testes para a versão 6 do standard, que formam 20% do total dos testes. Esta percentagem é causada por terem sido introduzidos novas funções e campos aos objetos *String* e *RegExp*, com a introdução de 8 funções para o objeto *String* e 4 campos e funções para o objeto *RegExp*. As restantes quatro versões, versões 8, 9, 10 e 11, do standard formam cerca de 9% dos testes do grupo, divididos por percentagens a rondar os 1,6% dos testes para as versões 9, 10 e 11, com a versão 8 a rondar os 3% dos testes. O grupo contém ainda 638 testes em que não foi obtida nenhuma versão do standard, o que representa uma percentagem a rondar os 24% dos teste do grupo.

Segue-se o grupo *Indexed Collections* que engloba os objetos *Array* e *TypedArray*, introduzidos nas versões 5 e 6 respectivamente. O grupo é composto por 4620 testes dos quais 2156 (47% dos testes) são referentes à versão 5 do standard que se devem aos testes referentes ao objeto *Array*, e aos seus campos e funções iniciais. Por sua vez, o grupo possui 1375 testes (30%) para a versão 6 do standard onde foi introduzido o objeto *TypedArray*, juntamente com as suas 26 funções iniciais e os seus 6 campos. Os testes do grupo *Indexed Collection* têm ainda 13% dos testes para a versão 9 do standard, o que corresponde a 618 testes, apesar de não ter sido introduzido nenhum objeto, função ou campo para este grupo. Os testes para as versões 8, 10 e 11 do standard fazem menos de 5% do total dos testes do grupo, que correspondem na grande maioria a testes para os objetos *Array* e *TypedArray* que utilizem objetos introduzidos nessas versões. O grupo têm ainda 260 testes que não foram filtrados para nenhuma versão, que corresponde a menos de 6% dos total de testes do grupo.

Na distribuição das versões obtidas na analise do grupo *Keyed Collection* observa-se que todos os testes em que foi encontrada uma versão corresponde à versão 6 do standard. Esta versão introduziu os objetos *Map*, *Set*, *WeakMap* e *WeakSet*, que são os objetos que formam o grupo *Keyed Collection*. Estes grupos não tiveram nenhuma adição de funções ou campos nas versões seguintes. Adicionalmente, uma pequena percentagem de testes(1%) não foi encontrada nenhuma versão.

O grupo *Structured Data* apresenta aproximadamente 8% dos testes com versão 5, testes que correspondem ao objeto JSON. O grupo têm uma percentagem de 35% de testes para a versão 6, que se devem a ter sido introduzido o objeto *ArrayBuffer* e *DataView* que fez aumentar o número de testes do grupo. Para a versão 8 existe também uma grande percentagem de testes com 23% dos testes, este aumento é causado pela estreia no standard dos objetos *SharedArrayBuffer* e *Atomics*. O grupo é apresenta ainda uma grande percentagem(24%) de testes em que não foi possível calcular a versão

do mesmo. Os restantes 10% dos testes estão repartidos pelas restantes versões do standard.

O grupo *Control Abstraction*, constituido pelos objetos *Iterator*, *AsyncIterator*, *GeneratorFunction*, *AsyncGeneratorFunction*, *Generator*, *AsyncGenerator*, *Promise* e *AsyncFunction*, têm a maioria dos testes associados à versão 6 do standard. Esta associação é devido a nesta versão ter sido a introdução dos objetos *Iterator*, *GeneratorFunction*, *Generator* e *Promise*, com a representação de 48% dos testes do grupo. A versão 8 do standard introduziu para este grupo o objeto *AsyncFunction* daí ter haver uma pequena percentagem dos testes (2%) para a versão 8. Para a versão 9 existem 6% dos testes dos grupos referentes à introdução dos objetos *AsyncGenerator* e *AsyncGeneratorFunction*. Existem ainda uma pequena percentagem de testes (5%) filtrados para a versão 11 do standard, esta versão não introduziu novos objetos, funções ou campos para o grupo. Este grupo incluí ainda uma grande percentagem de testes em que não foi possível encontrar a versão dos mesmos, atingindo os 38% de testes sem versão.

O grupo *Reflection* composto pos dois objetos, *Proxy* e *Reflection*, que foram introduzidos na versão 6 do standard, bem como todas as suas funções e campos, tem cerca de 89% dos seus testes a pertencer à versão 6. Este grupo possuí ainda 10% de testes que não foi encontrada nenhum versão, e os restantes 1% dos testes divididos pelas outras 4 versões.

O número de testes por versão do grupo *URI*, que engloba todos os objetos que tratem da codificação e descodificação de URIs. Através da analise do gráfico verificamos que a grande maioria dos testes faz parte da versão 5 do standard com cerca de 91% dos testes, o que corresponde a 156 testes num total de 168. O grupo têm 12 testes pertencentes à versão 6 do standard e uns meros 4 testes que não foram filtrados por nenhuma versão.

A distribuição das versões do grupo *statements* apresenta um grande número de testes referentes à versão 6 do standard, que introduziu os *statements Async Function*, *let* e *const*. Os dois últimos bastante prevalentes nos testes elevando o número de testes para esta versão. De forma que há 3003 testes para a versão 6 do standard que representa um terço dos testes do grupo. O grupo apresenta ainda das maiores percentagens de testes em que não foi possível encontrar versão, com 49% dos testes. Os restantes testes distribuem-se pelas outras versões com maior número de testes para as versões 5 e 10 do standard, com 9% e 6% dos testes respectivamente. A versão 11 apresenta o menor número de testes com apenas 20 testes neste grupo, o que traduz em 0,2% dos testes.

No grupo *expressions* destaca-se o grande número de testes que não foi encontrada nenhuma versão. O número de testes sem versão é de 4785 testes, que se traduz em 46% dos testes do grupo. Os testes do grupo para a versão 5 do standard são 1651, o que representa cerca de 16% dos testes do grupo a segunda versão com mais testes do grupo. A versão com mais testes deste grupo é a versão 6 com 2771 testes, o que equivale a 26% do total de testes. Os grupos que se seguem com maior número de testes são as versões 9 e 10, com um total de 9% entre as duas versões. As duas restantes

versões, a versão 8 e 11, correspondem a apenas 3% dos testes onde a versão 8 ocupa a maior fatia destes 3%.

Por fim, o grupo *other* que engloba todas as outras pastas do directório *language* apresenta um grande número de testes que não obtiveram versão na análise. Os testes sem versão representam 40% dos testes com 1192 testes. Os testes com versão 5 deste grupo ocupam 38% do total de testes, sendo a versão com maior número de testes. A versão 6 do standard representa 15% dos testes com 459 testes no grupo *other*. Os restantes 7% dos testes do grupo está distribuido pelas outra 4 versões, com a versão 9 a ter o menor número de testes com apenas 14.

Concluindo, a análise das versões para os testes da bateria de testes *Test262* permite-nos ter uma ideia geral do número de testes para cada versão do standard. Para os grupos do directório *built-in* podemos observar que as versões com maior número de testes são as versões onde foram introduzidos objetos, funções e/ou campos. O que se traduz as versões 5 e 6 terem mais testes, uma vez que estas são as versões que introduziram o maior número de objetos, funções e campos. Para os grupos no directório *language* apresentam um grande número de testes que não obtiveram nenhuma versão no decorrer da análise. Os grupos *expressions* e *statements* são caracterizados por terem um grande número de testes para a versão 5 e 6 do standard, com maior inclinação para a versão 6 devido a serem as versões que introduziram a maioria das contruções sintáticas e *built-ins*.

4.5 Built-ins nos Testes

Ao analisar os *built-ins* utilizados nos testes foram construídos gráficos circulares com os *built-ins* mais utilizados para todos os testes, para os testes pertencentes ao directório *built-ins* e ao directório *language*. Os gráficos representam o número de testes em que cada *built-in* aparece, podendo haver testes que não têm nenhum *built-in* associado e testes com vários *built-in*. Estes *built-ins* estão apresentados por ordem de maior presença, da esquerda para a direita e de cima para baixo. De forma a facilitar a observação dos testes com menor presença todos os *built-ins* que apresentam menos de 1% do total dos *built-ins* estão apresentados num segundo gráfico. Ao analisar os gráficos conseguimos ter uma ideia dos *built-ins* mais utilizados, notando ainda que estes resultados provêem da analise estática executada de forma que muitos testes não foi possível retirar os *built-ins* usados devido a falha no esprima ou a esperarem execuções com resultado de erro.

O gráfico da Figura 4.4 apresenta os *built-ins* mais utilizados no total dos testes, apresentando o número de ocorrências de cada *built-in*. Como podemos observar os *built-ins* mais utilizados são o *Object* e o *Array*, pois para além dos testes para os próprios *built-in* existem muitos testes que os utilizam de forma a testar certos comportamentos. O terceiro *built-in* mais frequente na bateria de testes é o objecto *Error* que é usado em vários testes para confirmar que um certo comportamento retorna erro.

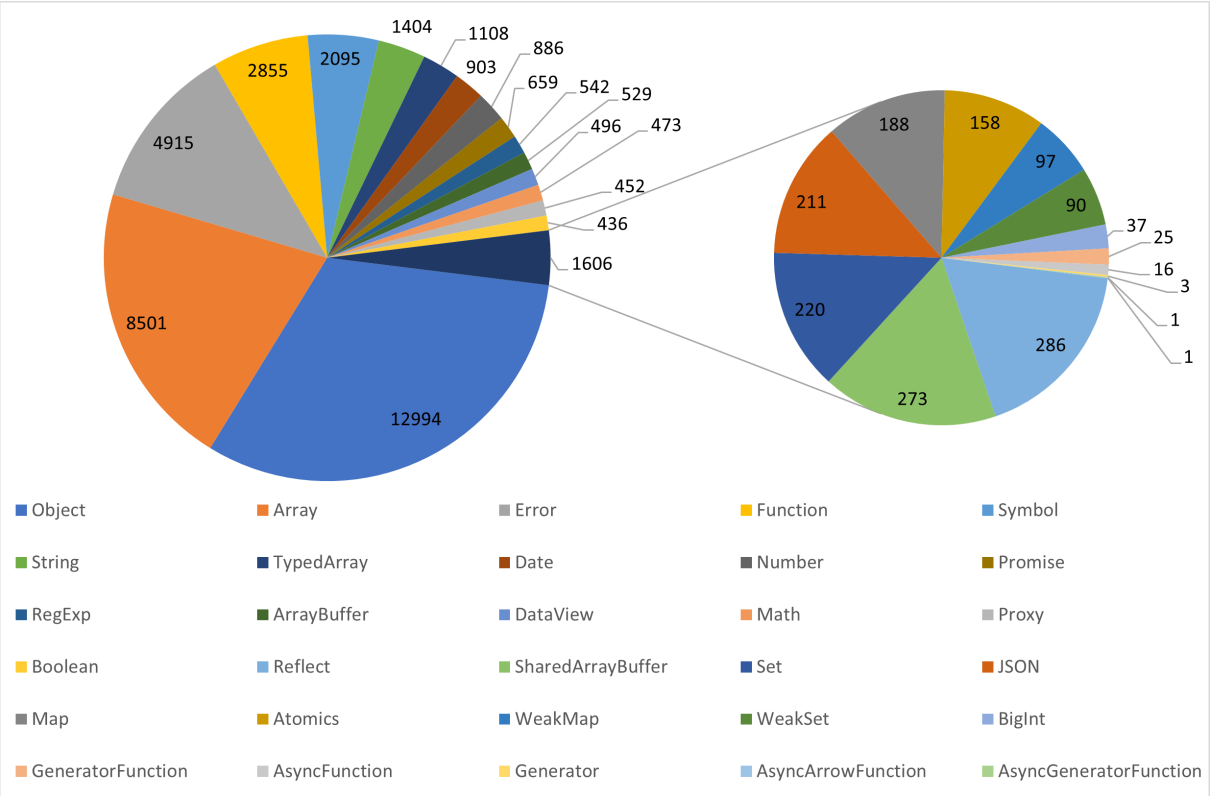

Figura 4.4: Número Total dos Objetos na Test262

Os três objectos que se seguem com maior presença nos testes são os objectos *Function*, *Symbol* e *String* todos objectos do grupo *fundamental* que vão ocupar 7%, 5% e 3% dos testes, respectivamente. Cerca de 4% dos *built-ins* encontrados são repartidos pelos objectos que representam menos de 1% dos testes, que vão ser 14 objectos. Numa análise mais generalizada concluímos que os objectos mais usados pertencem ao grupo *fundamental*, pois ao serem objectos fundamentais para a execução da linguagem JavaScript vão aparecer num grande número de teste de forma a testar a auxiliar a testagem de diversos comportamentos. Do gráfico apresentado retiramos ainda que os *built-ins* menos utilizados são os objectos introduzidos mais recentemente, os objectos asincronos e os objectos relacionados com o *Generator*. O número reduzido de ocorrências deve-se as limitações do esprima que não suporta os objectos *assincronos* e os *Generator*, a abordagem dinâmica também não apresenta melhores resultados.

De seguida, analisámos o número de vezes que os objetos aparecem para os testes específicos desses objetos para isto foi construído o gráfico da Figura 4.5. Onde se retira que os objectos mais utilizados nos testes vão ser os objectos *Object*, *Array* e *TypedArray*. Os objetos *Object* e *Array* têm uma grande representação por serem objetos fundamentais e com muitas funções e campos. O objetos *TypedArray* aparece por todos os testes deste *built-in* aparecem em testes específicos. Os objectos que

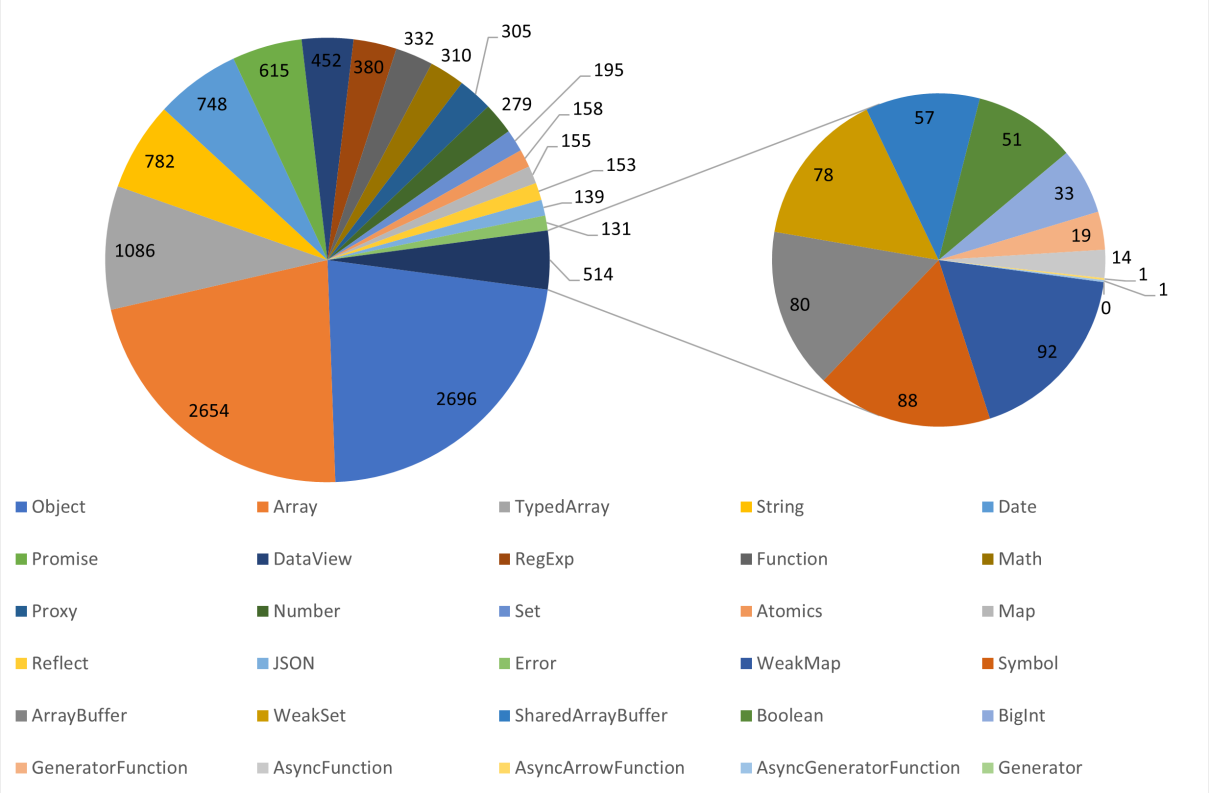

Figura 4.5: Objetos para Testes Específicos do Objeto

se seguem com maior relevância nos testes do directório são os objectos *String* e *Date* que formam quase 13% do total de *built-ins* encontrados. O objecto *Promise* aparece a seguir com 5% dos objectos encontrados. O gráfico demonstra ainda que existem 12 objectos que representam menos de 1% dos *built-ins* encontrados no directório *built-in*. Mais uma vez, os objectos assincronos e *Generators* encontram-se em número reduzido devido à análise realizada falhar na grande maioria dos testes para estes objectos. Por fim, concluímos que os objectos *Object*, *Array* e *TypedArray* encontram-se em grande número todos eles com mais de 9% do gráfico. No entanto, existe um grande número de objectos (12) que não ultrapassa os 1% do total dos objectos.

O número de *built-ins* que aparecem nos testes não específicos do *built-in* estão representados no gráfico da Figura 4.6, onde aparece no primeiro gráfico circular os objectos que compõe mais 1% do gráfico. Enquanto que no segundo grupo apresenta todos os restantes objetos com representação de menos de 1% do gráfico. Os objectos mais comuns são, semelhantemente ao total da bateria de testes os objectos *Object*, *Array* e *Error*. Estes três objectos juntos são 73% do gráfico. Para além dos três objectos mais usados apenas mais 6 objectos representam mais de 1% dos *built-ins*, esses objectos são o *Function*, o *Symbol*, o *String*, o *Number*, o *ArrayBuffer* e o *Boolean* todos objectos pertencentes ao grupo *fundamental*. Existem 17 objectos com representação de menos de 1% dos

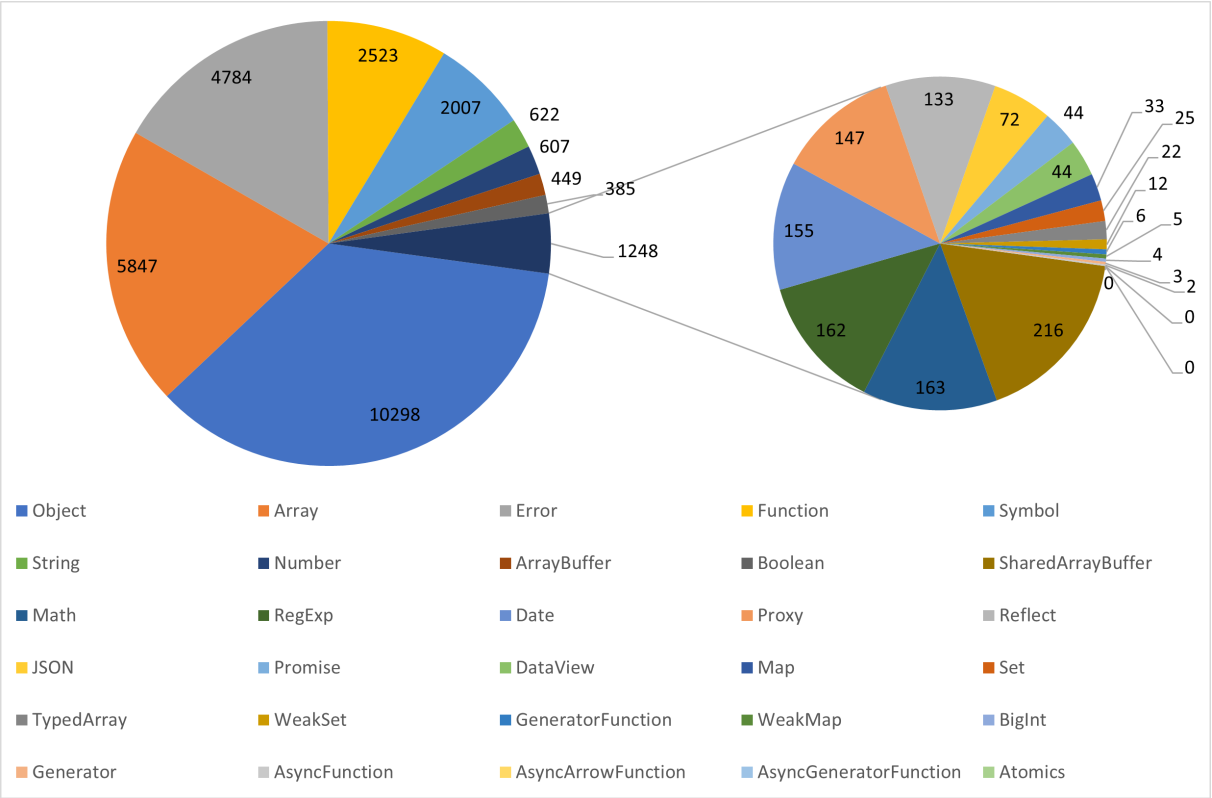

Figura 4.6: Objetos para Testes não Específicos do Objeto

built-ins, todos eles juntos apenas representando 4% do total dos *built-ins*. Existem ainda 3 *built-ins* que não aparecem em nenhum testes não específico do *built-in*, estes objectos são *AsyncArrowFunction*, *AsyncGeneratorFunction* e *Atomics*.

De seguida, foi encontrado o número de vezes que cada *built-in* aparece em testes não específicos para o *built-in*. Na Tabela 4.5 podemos observar o número de testes juntamente com a percentagem das ocorrências dos *built-ins* que aparecem em testes não específicos para o *built-in*. Como é possível observar o objecto com maior percentagem de testes fora dos testes específicos do *built-ins* é o *Generator*, com 100%. Estes números devem-se à maioria dos testes com a ocorrência do *Generator* o esprima não ser capaz de os analisar. Os *built-ins* que se seguem com uma percentagem alta de testes fora dos testes específicos são os objectos que fazem parte do grupo *fundamental*, mais especificamente os objectos *Object*, *Function*, *Symbol*, *Boolean* e *Error*. Para além dos objectos do grupo *fundamental* apresentam ainda percentagens altas os objectos *Number* e *Array* também muito presentes nos grupos *statement* e *expressions* do directório *language*. Ainda com alta percentagem aparecem os objectos *ArrayBuffer* e *SharedArrayBuffer* que vão aparecer em número elevado nos testes do objecto *DataView*. Os objectos que se seguem com maior percentagem, com percentagens entre os 25% e os 50%, são os objectos *Reflect*, *String*, *Math*, *JSON*, *Proxy* e *RegExp*. Estes são objectos que ainda

aparecem com alguma frequência nos testes não específicos dos mesmos mas com a sua maioria dos testes a pertencer aos testes específicos do objectos. Os restantes *built-ins* aparecem quase na totalidade nos testes específicos dos mesmos, havendo a exceção dos objectos *AsyncArrowFunction* e *AsyncGeneratorFunction* que são objectos que esta análise não contempla.

Tabela 4.5: Percentagem dos Objetos em Testes não Específicos

Built-Ins	Nº de Testes	Percentagem
Generator	3	100,00%
Error	4784	97,34%
Symbol	2007	95,80%
Function	2523	88,37%
Boolean	385	88,30%
ArrayBuffer	449	84,88%
Object	10298	79,25%
SharedArrayBuffer	216	79,12%
Array	5847	68,78%
Number	607	68,51%
Reflect	133	46,50%
String	622	44,30%
Math	163	34,46%
JSON	72	34,12%
Proxy	147	32,52%
RegExp	162	29,89%
GeneratorFunction	6	24,00%
Map	33	17,55%
Date	155	17,17%
WeakSet	12	13,33%
AsyncFunction	2	12,50%
Set	25	11,36%
BigInt	4	10,81%
DataView	44	8,87%
Promise	44	6,68%
WeakMap	5	5,16%
TypedArray	22	1,99%
AsyncArrowFunction	0	0,00%
AsyncGeneratorFunction	0	0,00%
Atomics	0	0,00%

5

Sistema de apoio à testagem

Conteúdo

No decorrer desta tese foram criadas duas bases de dados com o intuito de auxiliar no desenvolvimento de implementações do standard da linguagem ECMAScript. Ambas as bases de dados são guardadas na base de dados MongoDB, uma base de dados noSQL que permite introduzir dados com formato JSON. A primeira base de dados vai conter metadata sobre os testes, a metadata dos testes é composta pela metadata já presente nos teste, bem como a obtida pelos métodos descritos no Capítulo 3. Esta base de dados vai permitir filtrar testes pela metadata desejada, como por exemplo obter apenas os testes referentes à versão 5 do standard ou um exemplo mais complexo obter os testes que possuam os *built-ins Array* e *Function*, e as construções sintáticas *ExpressionStatement* e *CallExpression*.

A segunda base de dados criada têm como objetivo armazenar os resultados de execuções dos testes, guardando o resultado final dos testes, o seu tempo de execução entre outros dados sobre a execução. Esta base de dados pretende acelerar a análise dos resultados e comparar os resultados com execuções passadas. Esta análise permite obter os testes que cumpram com a filtragem pedida, tanto filtragem pelo resultado como pelo tempo de execução.

Para auxiliar no uso das bases de dados foram desenvolvidas para cada uma das bases de dados uma aplicação de terminal. Esta aplicação permite a utilizadores que não possuam conhecimento sobre as queries de MongoDB utiliar as bases de dados. As próximas duas secções descrevem em maior detalhe as bases de dados desenvolvidas, bem como as aplicações desenvolvidas mostrando o seu comportamento. Começando pela base de dados destinada à análise da metadata dos testes, seguido da base de dados para os resultados das execuções dos testes.

5.1 Base de Dados de Testes

A primeira base de dados criada foi criada no âmbito de guardar toda a metadata dos testes. A metadata relativa a cada teste é armazenado em formato JSON. O formato dessa metadata está representado na Figura 5.1 para o teste ”15.4.5-1.js” presente no directório ”test/built-ins/Array”. A metadata que é guardada consiste na já presente no teste e previamente descrita, bem como a metadata adicional calculada na secção acima:

- *version* - Representa a primeira versão do standard para a qual o teste vai pertencer, caso não tenha sido possível calcular a versão este campo não é apresentado;
- *syntactic_construct* - Um array composto por todas as construções sintáticas presentes na arvóre sintática dos testes;
- *builtIns* - Um Array com todos os built-ins existentes no teste;
- *asserts* - Número de chamadas ”assert” que existem no teste;

```
{
    "path": "./test262-main/test/built-ins/Array/15.4.5-1.js",
    "version": 5,
    "esid": " 15.4.5-1",
    "description": " Array instances have [[Class]] set to 'Array'",
    "built-ins": "Array",
    "Array": "15.4.5-1.js",
    "syntactic_construct": [
        "Identifier",
        "ArrayExpression",
        "VariableDeclarator",
        "VariableDeclaration",
        "MemberExpression",
        "CallExpression",
        "Literal",
        "ExpressionStatement",
        "Program"
    ],
    "builtIns": {
        "Array": [],
        "Object": [
            "prototype",
            "prototype.toString"
        ],
        "Function": [
            "call"
        ]
    },
    "asserts": 1,
    "error": 0,
    "esprima": "supported",
    "lines": 3
}
```


Figura 5.1: Estrutura do JSON para a nossa proposta de metadata

- *errors* - Número de chamadas ”$ERROR” presentes no teste;
- *lines* - Número de linhas de código do teste, este número exclui linhas de comentário e linhas em branco;

Adicionalmente, a metadata contém campos referentes aos directórios que o teste pertence, começando nos directórios na pasta ”test262-main/test/”. Assim sendo, para a metadata exemplo da Figura 5.1 com *path* de ”./test262-main/test/built-ins/Array/15.4.5-1.js”a metadata contem os campos ”built-ins”com valor associado de ”Array”e ”Array”com valor associado de ”15.4.5-1”.

Com a metadata no formato JSON é agora possível inserir diretamente na base de dados noSQL, MongoDB. A partir da base de dados podemos filtrar os testes pela metadata, através do terminal pelo MongoDB, do GUI do MongoDB (MongoDB compass) ou da aplicação terminal produzida no âmbito desta tese.

As queries suportadas pela base de dados permitem ao utilizador filtrar os testes por versão, por construção sintática presente ou não no teste, por built-ins presente ou não no teste e/ou pelo número de linhas, *asserts* e/ou *errors*. É também possível filtrar os testes que contêm ou não a metadata. Para a realização das queries é usado o formato JSON. Por exemplo, para filtrar os testes que pertençam à versão 5 e contenham ambas as construções sintáticas *"ArrayEspression"* e *"CallExpression"* usando o terminal MongoDB e no GUI do MongoDB:

```
{ "version" : 5, "$and" : [ { "syntactic_construct" : "ArrayExpression" }, {
"syntactic_construct" : "CallExpression" } ] }
```

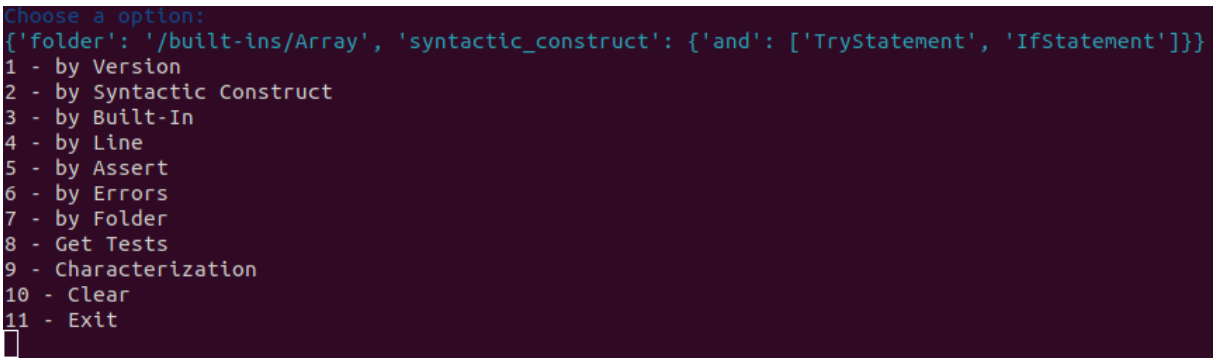

Figura 5.2: Menu principal da aplicação da metadata

A aplicação foi criada para facilitar as consultas à base de dados eliminando a necessidade de aprender a executar as queries Mongodb. Ao correr a aplicação vai ser apresentado um menu principal que é visível na Figura 5.2. O menu contém a azul claro o estado da query atual, na Figura 5.2 o estado da query é os testes no diretório "test/built-ins/Array" cuja sua árvore sintática contenha *TryStatement* e *IfStatement*. A cor branco as opções para filtragem e apresentação de resultados:

1. *by Version* - filtra os testes por versão, podendo filtrar por mais de uma versão em cada consulta;
2. *by Syntactic Construct* - com opção de filtrar testes que contenham todas as construções sintáticas escolhidas ou os testes com contenham no minimo uma das construções sintáticas escolhidas;
3. *by Built-In* - uma vez escolhida, pede ao utilizador para escolher se quer que a filtragem devolva os testes que apresentem todos os *built-ins* escolhidos ou os testes que apresentem pelo menos um dos *built-ins* selecionados;
4. *by Line* - possibilita filtrar testes pelo seu número de linhas, escolhendo os testes que têm mais linhas ou os testes que têm menos linhas que o número selecionado;
5. *by Assert* - permite escolher um número de asserts para filtrar os testes com mais ou menos asserts que o número escolhido;

6. *by Errors* - fornece a opção de escolher os testes com menos ou mais *errors* presentes nos testes do que o número selecionado;
7. *by Folder* - filtra os testes por directórios escolhido, permitindo considerar apenas os testes do directório selecionado
8. *Get Tests* - guarda os testes num ficheiro com nome sugestivo à query executada;
9. *Characterization* - imprime no terminal o número de teste, a média das linhas dos testes, a média de *asserts* presentes nos testes e a média de *errors* presentes nos testes referentes à consulta;
10. *Clear* - limpa o estado atual da query;
11. *Exit* - fecha a aplicação.

A Figura 5.3 apresenta o menu da aplicação quando a opção *by Version* é escolhida. Após a seleção desta opção são apresentadas as opções para o estado atual da query, a branco, e a indicação de como registar as versões pela qual se quer filtrar, a azul claro. A escolha pode ser de apenas uma versão ou de mais que uma versão. Caso seja mais do que uma versão têm de ser separadas por virgula. A opção *by Folder* tem um comportamento semelhante, apresentando todas as pastas dentro do directório atual que o utilizador pode escolher apenas uma pasta.

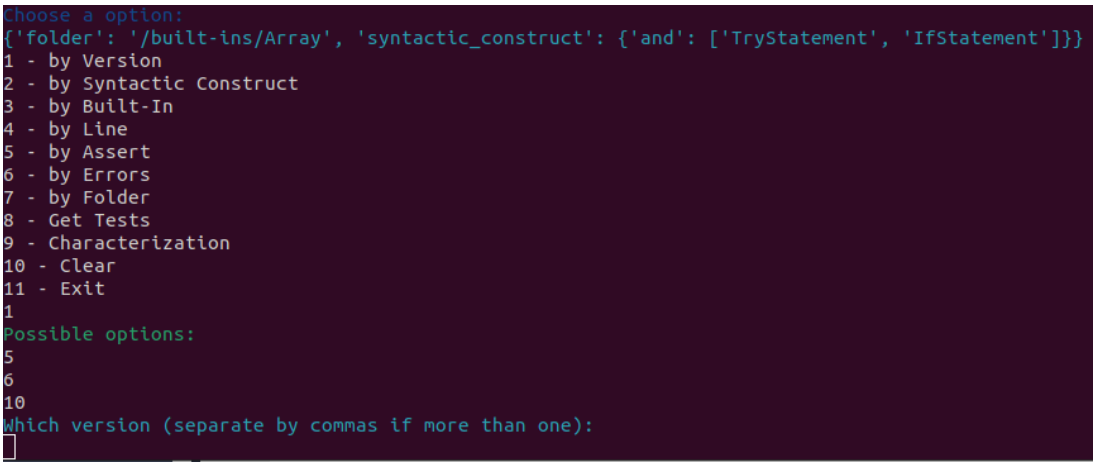

Figura 5.3: Aplicação da metadata quando a opção 1 é escolhida

Para as opções *by Syntactic Construct* e *by Built-In* a aplicação começa por perguntar se o utilizador quer que a escolha seja em união ou em intersecção, *and* ou *or* respetivamente. De seguida, são apresentadas as opções disponíveis para o estado atual da pesquisa, a branco, podendo ser escolhidas várias opções, as mesmas sempre separadas por virgulas. Este comportamento é observado na Figura 5.4, que apresenta o comportamento da aplicação para a opção número 3, *by Built-In*, onde já foi escolhida a opção de intersecção das opções.

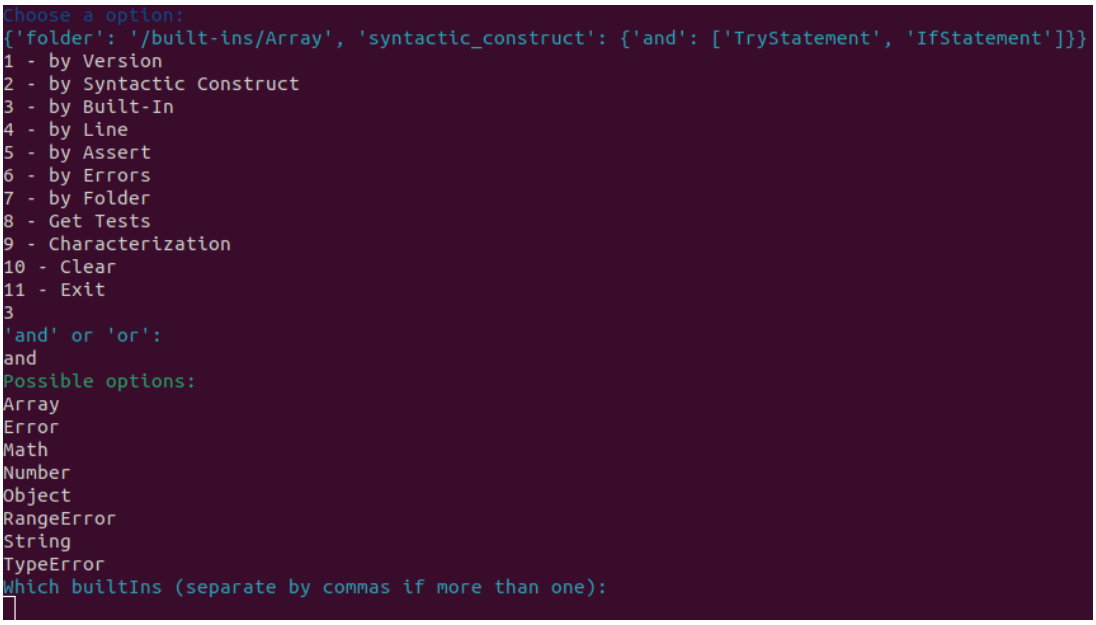

Figura 5.4: Aplicação da metadata quando a opção 3 é escolhida

Por sua vez, ao escolher as opções *by Line*, *by Asserts* e *by Errors* a aplicação pede ao utilizador se quer filtrar por mais ou por menos do que a opção escolhida. So faltando registar o número pelo qual é filtrado, obtendo todos os testes que se cumpram o filtro escolhido. Como pode ser observado na Figura 5.5 após a escolha da opção *by Line* é pedido se quer filtrar por mais ou menos, *more or less asserts* a azul claro, seguido pelo número de asserts que se quer filtrar.

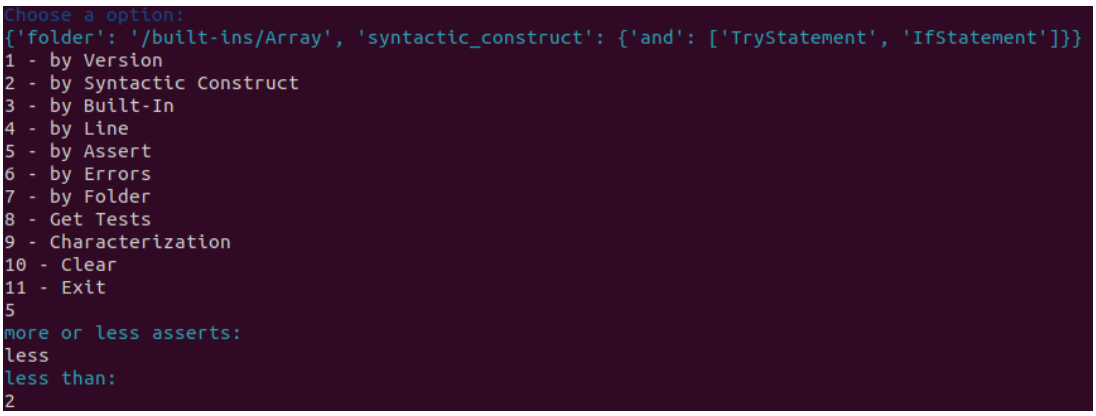

Figura 5.5: Aplicação da metadata quando a opção 5 é escolhida

Ao escolher a opção *Characterization* são impressos na consola o número de testes que a query atual retorna, assim como o número médio de linhas, asserts e erros. A Figure 5.6 demostra este comportamento onde o texto a verde indica as caracteristicas da query atual.

Por fim, ao selecionar a opção *Get Tests* são gravados os resultados da query atual armazenando os resultados num ficheiro com nome sugestivo à query atual. Os testes são guardados numa pasta

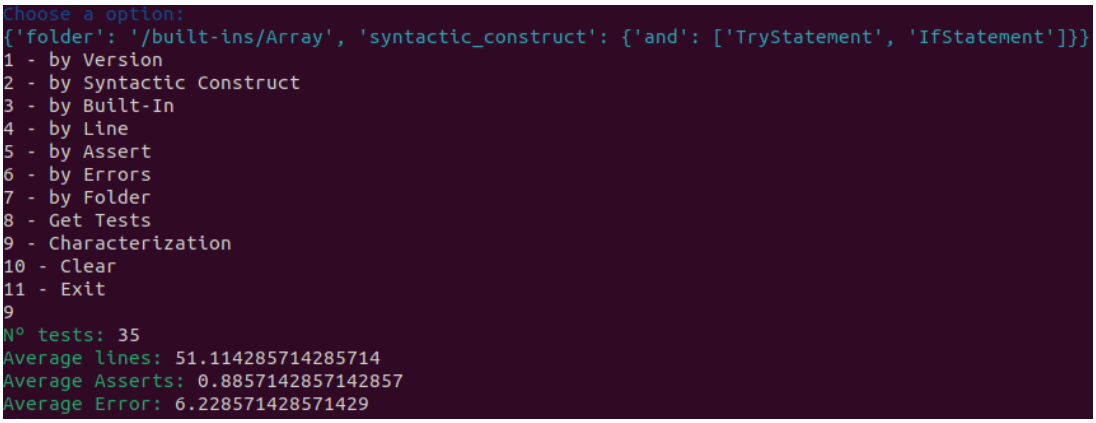

Figura 5.6: Aplicação da metadata quando a opção 9 é escolhida

`Selected_Tests` com o nome começado por `SELECTION` seguido pelos filtros separados por -". Este comportamento é visível em baixo para a query que têm sido usado nos exemplos, isto é filtrar os testes do directório e para as construções sintáticas *TryStatement* e *IfStatement*. As restante opções, *Clear* e *Exit*, limpam o estado atual da query e sair da aplicação respetivamente.

```
./Selected_Tests/SELECTION-built-ins-Array-syntactic_construct-TryStatement_and_IfStatement.txt
```

5.2 Base de Dados de Resultados


```
{
    "test_id": 1636372444741,
    "File path": "test/test262/tests/language/punctuators/S7.7_A2_T5.js",
    "Result": "_OK_",
    "Observations": "",
    "Creation of the AST": "",
    "Plus to Core": 0.097,
    "Interpretation": 0.133,
    "tests": "language",
    "language": "punctuators",
    "punctuators": "S7.7_A2_T5.js"
}
```


Figura 5.7: Estrutura do JSON para apresentação dos resultados

A segunda base de dados tem como objetivo armazenar os resultados das execuções dos testes e comparar os mesmos com execuções passadas. Mais uma vez foi utilizada a base de dados MongoDB, por isso os resultados foram transformados para o formato JSON. O JSON criado para um teste exemplo está representado na Figura 5.7. O formato JSON contem como chaves:

- *File path* - caminho para o teste;
- *Result* - resultado da execução do teste;
- *Observations* - possíveis observações sobre os testes
- *Creation of the AST* - tempo de criação da AST para a realização dos testes (opcional, caso a AST já esteja criada previamente a execução dos testes)
- *Plus to Core* -
- *Interpretation* - tempo de execução do teste

Como na base de dados de testes, esta segunda base de dados também contém chaves para as sub-pastas dos testes começando no directório ”test/test262main/tests”. Como pode ser visualizado no exemplo da Figura 5.7 nos três últimos campos, onde para o teste com caminho ”test/-test262main/tests/language/punctuators/S7.7_A2_T5.js” vai conter as chaves para o ”tests”, ”language” e ”punctuators” e os valores correspondentes de ”language”, ”punctuators” e ”S7.7_A2_T5.js”.

O ficheiro JSON criado contém um *Array* com o objeto com a informação para cada teste. Semelhantemente à base de dados da metadata agora podemos adicionar o ficheiro JSON criado à base de dados. Com a informação inserida na base de dados é possível agora consultar e filtrar os resultados, através do terminal ou do GUI do MongoDB, MongoDB compass.

A base de dados permite filtrar os resultados, sendo possível alcançar o número de teste que passaram, que falharam ou que deram erro, calcular o tempo total de execução dos testes, tempo médio de execução, tempo máximo ou minímo dos testes de uma determinada filtragem. Sendo ainda possível filtrar os testes por directório e todas as combinações das consultas acima descritas. O exemplo de uma query é apresentado abaixo, a query pretende filtrar os testes pertencentes ao directório ”language”, que passaram com sucesso e com tempo de interpretação inferior a 0.1 segundos.

```
{ tests : "language", Result : "_OK_", Interpretation : { "$lt" : 0.1 } }
```

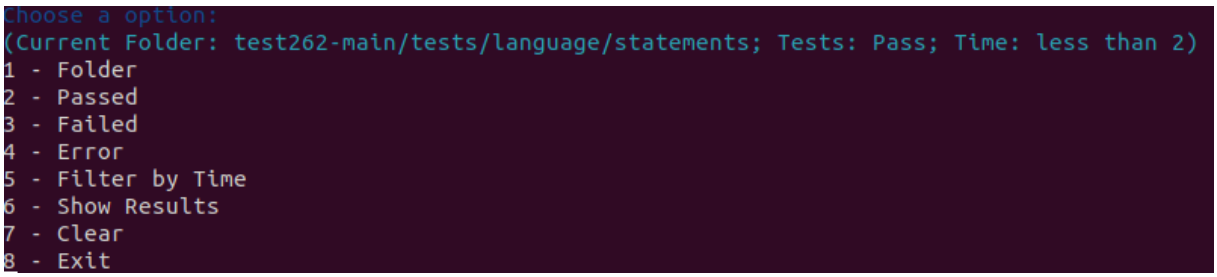

Figura 5.8: Menu principal da aplicação dos resultados

Adicionalmente, foi criada uma aplicação python destinada ao auxílio da consulta removendo a necessidade de aprender os comandos de consulta do MongoDB. O menu principal da aplicação criada

é apresentado na Figura 5.8, como pode ser observado o menu principal é semelhante a aplicação para filtragem da metadata dos teste. Contendo a azul claro o estado da consulta e branco as opções de filtragem. As opções de filtragem possiveis vão ser:

1. *Folder* - Permite escolher o directório para o qual a consulta vai ser executado;
2. *Passed* - Filtra os testes que tiveram execução correta;
3. *Failed* - Filtra os testes que retornaram errado na execução desejada;
4. *Error* - Filtra os testes que retornaram *Error* na execução;
5. *Filter by Time* - Fornece a opção de filtrar os testes que demoram mais/menos tempo que o limite proposto a executar;
6. *Show Results* - Imprime os resultados para um ficheiro com nome sugestivo à consulta executada;
7. *Clear* - Limpa o estado da consulta atual;
8. *Exit* - Fecha a aplicação.

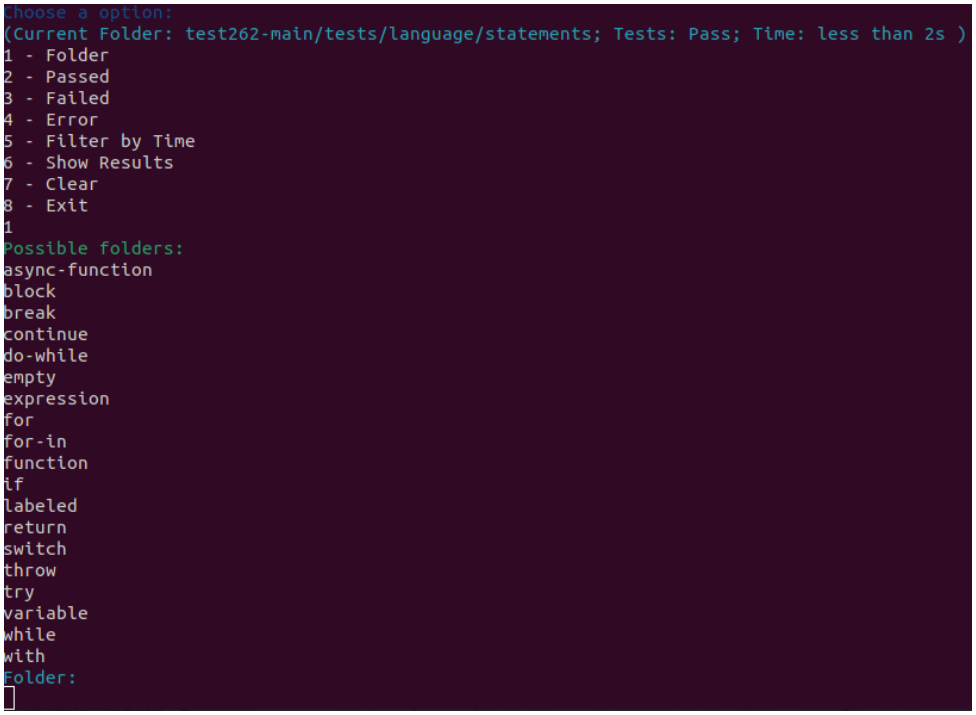

Figura 5.9: Aplicação da metadata quando a opção 1 é escolhida

Ao optar pela filtragem por directório, isto é ao selecionar a opção 1 da aplicação, a aplicação imprime no terminal todas as pastas presentes no directório atual. Após a listagem das pastas é pedido

ao utilizador que selecione a pasta que pretende entrar. Este comportamento está representado na Figura 5.9 onde pode ser observado o menu principal seguido da opção escolhida, neste caso ”1”. Após a seleção é impresso no terminal os directórios possíveis, apresentados depois da linha a verde *”Possible folders:”*. Por fim, temos a azul claro a linha *”Folder:”* que espera pela escolha do utilizador verificando se a opção é uma opção válida. Caso seja, o estado atual é atualizado e o menu principal é de novo apresentado.

```
Choose a option:
(Current Folder: test262-main/tests/language/statements; Tests: All; Time: less than 2s )
1 - Folder
2 - Passed
3 - Failed
4 - Error
5 - Filter by Time
6 - Show Results
7 - Clear
8 - Exit
2

Choose a option:
(Current Folder: test262-main/tests/language/statements; Tests: Pass; Time: less than 2s )
```

Figura 5.10: Aplicação da metadata quando a opção 2 é escolhida

Para as opções 2, 3 e 4 da aplicação o comportamento da mesma é semelhante. Onde após a escolha da opção o estado da aplicação muda automáticamente para a opção escolhida. A Figura 5.10 demostra aplicação quando é escolhida a opção 2, mostrando apenas os testes que passam, como é visto na alteração da linha do estado da aplicação, texto a azul claro, que passa de *”All”* para *”Pass”*.

```
Choose a option:
(Current Folder: test262-main/tests/language/statements; Tests: Pass; Time: s )
1 - Folder
2 - Passed
3 - Failed
4 - Error
5 - Filter by Time
6 - Show Results
7 - Clear
8 - Exit
5
more or less:
less
less than:
2

Choose a option:
(Current Folder: test262-main/tests/language/statements; Tests: Pass; Time: less than 2s )
```

Figura 5.11: Aplicação da metadata quando a opção 5 é escolhida

A opção de filtragem por tempo começa por perguntar se a filtragem é para mais ou menos do que o tempo escolhido, a azul claro a linha *”more or less:”*. O utilizador deve responder com *more* ou *less* para mais ou menos, respectivamente, seguido do tempo pelo qual quer filtrar. Depois de escolhida a opção

o estado da aplicação volta a ser atualizado possuindo agora no campo *Time* a opção de filtragem escolhida. O comportamento descrito esta presente na Figura 5.11.

Por fim as opções *Show Results*, *Clear* e *Exit* tem comportamentos semelhantes as opções com o mesmo nome da aplicação de filtragem da metadata. Onde a opção de *Show Results* guarda todos os testes que cumpram o estado atual da aplicação num ficheiro com nome sugestivo ao estado atual da aplicação. Este ficheiro é armazenado dentro de um directório com nome *"Queries-"* mais o nome do ficheiro dado ao inicializar a aplicação. Para o exemplo de filtragem dos testes que passaram dentro do directório *language/statements* da Test262 que demoraram menos de 2s a executar o nome que a aplicação vai corresponder é o apresentado em baixo. As opções *Clear* e *Exit* limpam o estado atual da aplicação e fecham a aplicação, respetivamente.

```
Pass-less_than_2-test262-main_tests_language_statements.txt
```

6

Avaliação

De modo a avaliar o trabalho desenvolvido no decorrer desta tese foram comparados os resultados obtidos no cálculo da versão dos testes com os testes filtrados manualmente para a versão 5 do standard para o projeto ECMARef. Podendo assim retirar conclusões da precisão do nosso calcúlo da versão dos testes. Para esse efeito foram construídos um diagrama e um gráfico que nos vão ajudar a interpretar os resultados obtidos. O diagrama, apresentado na Figura 6.1, compara todos os testes filtrados pelo nosso trabalho com os testes do ECMARef. Enquanto que o gráfico da Figura 6.4 apresenta a distribuição dos testes do ECMARef na nossa filtragem.

Ao observar o gráfico apresentado na Figura 6.1 podemos concluir que os nossos resultados associaram um número semelhante de testes à versão 5 do que a filtragem do ECMARef, havendo alguns testes que a filtragem do ECMARef associou à versão 5 do standard e que a nossa análise filtrou para versões diferentes. Os testes que foram filtrados pelo ECMARef e que na nossa filtragem obtiveram versões diferentes foram na maioria devido ao Node.js, havendo apenas 3 testes que são retirados da versão 5 pela análise estática. Por sua vez os testes que a nossa análise associou à versão 5 do standard e que não foram filtrados pelo ECMARef são na sua maioria relativos aos URI e a *escape characters*.

Dos 457 testes que a nossa filtragem associou à versão 5 do standard e que o ECMARef não considerou ser desta mesma versão podemos ver Figura 6.2 que cerca de 133 testes são relativos a URIs e que 111 são referentes a *escape characters*, ambas não consideradas no projeto ECMARef. Destacam-se ainda 45 testes relativos a separatores númericos e cerca de 30 testes que têm a tag indicadora da versão 5 do standard. Os restantes 138 testes são filtrados por motivos diversos

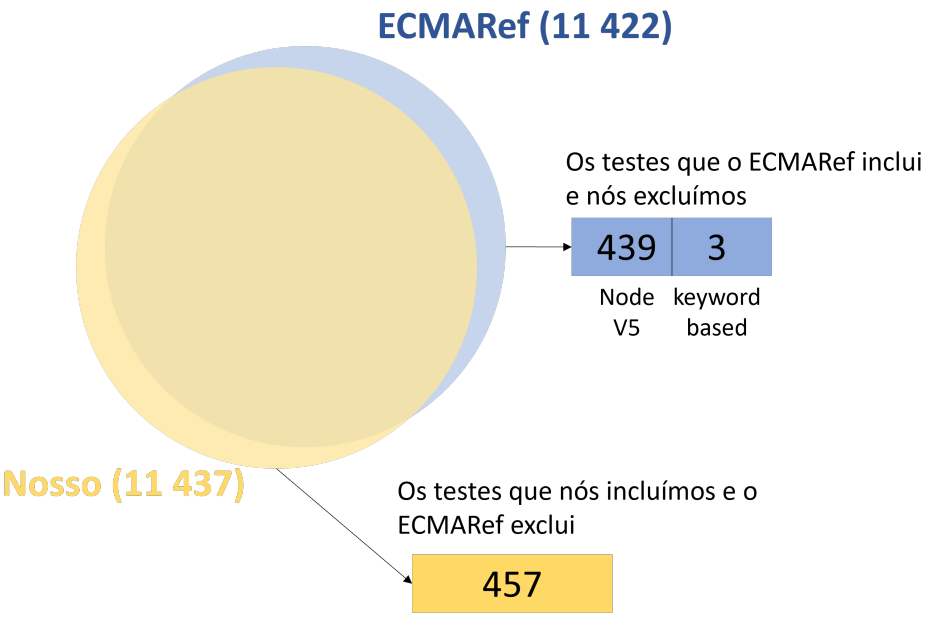

Figura 6.1: Comparação dos Nossos Resultados com os Resultados do ECMARef

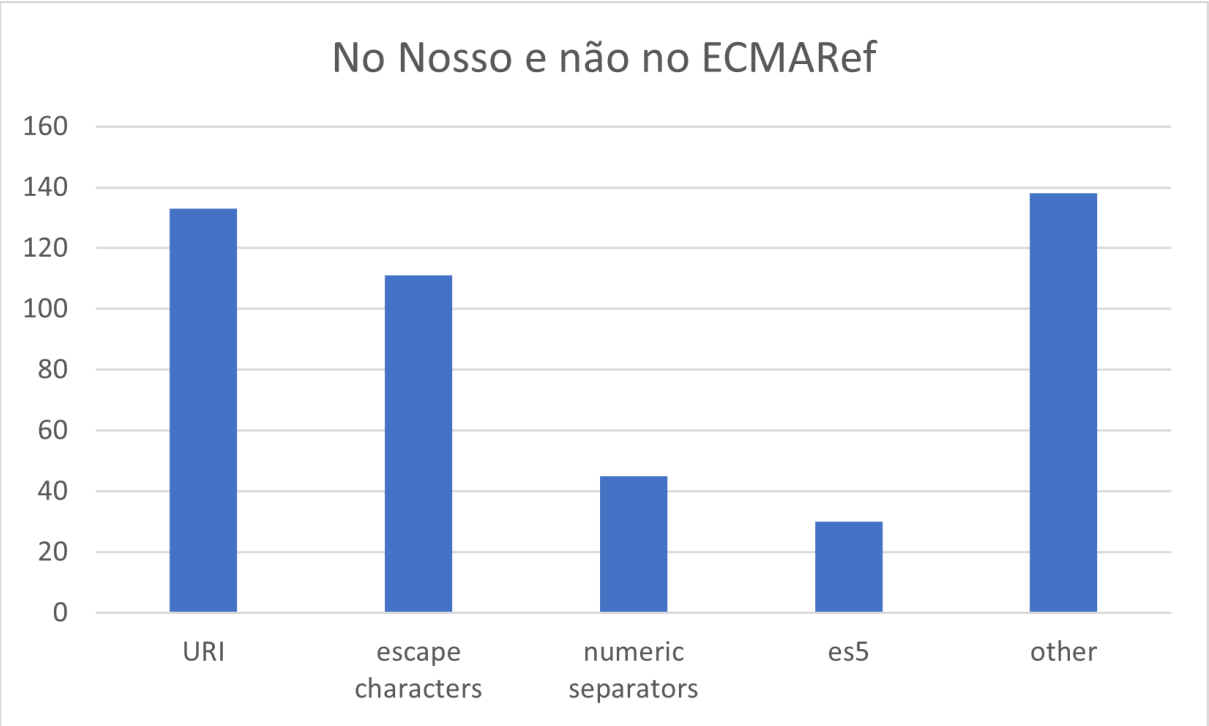

Figura 6.2: Distribuição dos testes que estão presentes na nossa filtragem e não foram pelo ECMARef

Dos 442 testes que o ECMARef filtrou para a versão 5 do standard e que a nossa filtragem não associou à versão 5 do standard mais de metade deve-se a uma implementação incorreta do `this` por parte do Node.js. É de notar ainda que 84 testes são devido aos atributos das propriedades. Estes números podem ser vistos na Figura 6.3

O gráfico da Figura 6.4 que representa a azul o número de testes filtrado pelo ECMARef, a laranja as versões que a análise dinâmica obteve e a cinzento as versões que a análise mista obteve. Concluíndo que existe uma concordância que a grande maioria dos testes pertence à versão 5 do standard e que os resultados da análise dinâmica são quase semelhantes aos da análise mista.

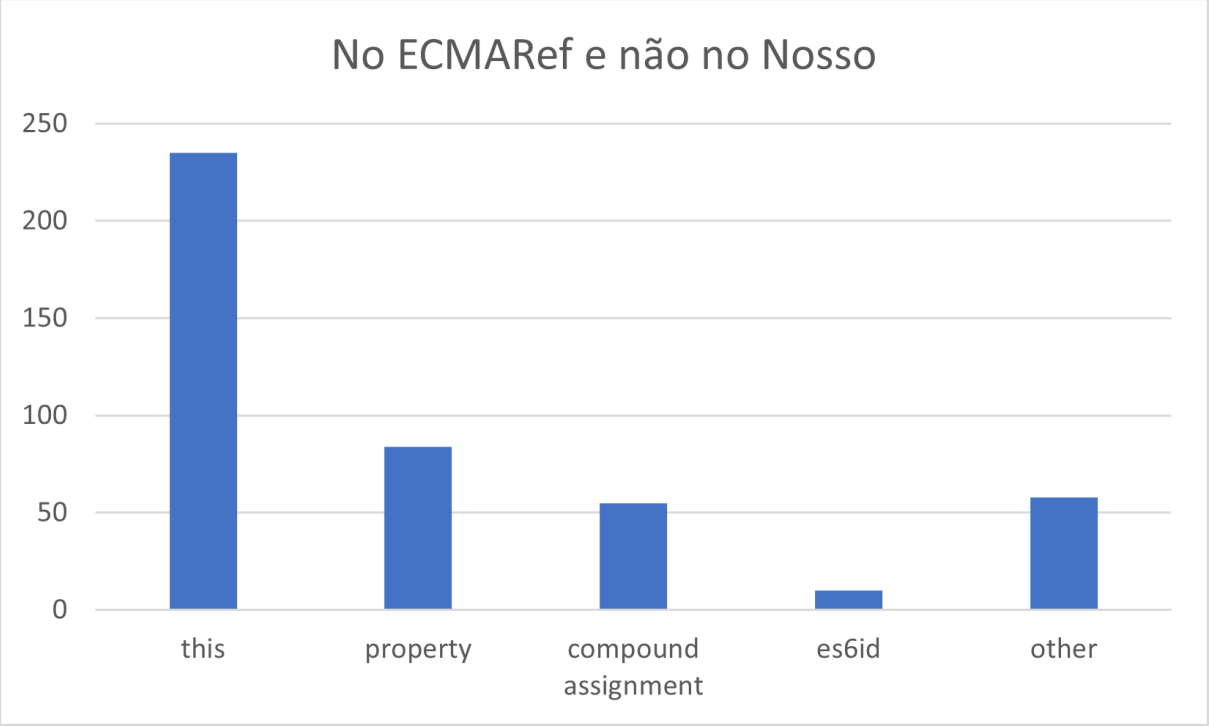

Figura 6.3: Distribuição dos testes que estão presentes no ECMARef e não foram filtrados pelo nosso

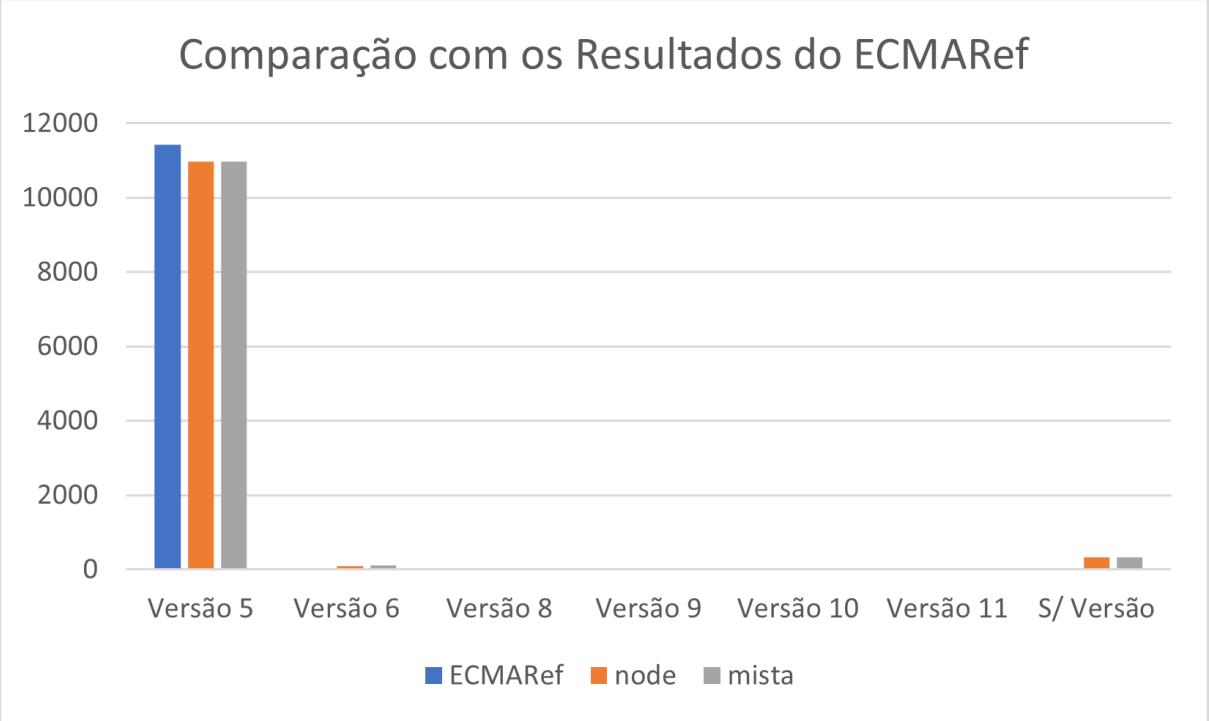

Figura 6.4: Comparação da nossa análise para os testes filtrados pelo ECMARef

7

Conclusão e Trabalho Futuro

Concluíndo o trabalho realizado no decorrer desta tese calculou metadata relevante para os testes da bateria oficial de testes da linguagem JavaScript. A metadata calculada engloba as construções sintáticas que compõe a árvore sintática do teste, a versão do standard à qual o teste pertence e os built-ins que aparecem nos testes juntamente com as suas funções e propriedades. Adicionalmente, a metadata calculada permite a caracterização da bateria de testes da Test262, esta caracterização permite ganhar uma maior compreensão sobre a Test262 e os testes que a compõem. O desenvolvimento tanto da aplicação de testes como da aplicação de resultados permite ignorar a aprendizagem de queries noSQL e focar na obtenção dos testes para a filtragem desejada.

Com a execução desta tese concluímos que a metadata dos testes atual na bateria oficial Test262 encontra-se bastante incompleta, oferecendo pouca informação sobre o teste a que é relativa. O que se traduz num aumento na dificuldade do processo de teste de implementações parciais do standard ECMAScript, não havendo nenhum processo de obter os testes para uma implementação parcial, com exceção de uma filtragem manual. O nosso trabalho fornece uma solução para este problema, embora imperfeita. O trabalho realizado complementa-se começando com o cálculo dos metadados dos testes e utilia os mesmos para a obtenção de testes para implementações parciais do standard. A metadata calculada permite uma maior compreensão do teste em questão e das funcionalidades que a implementação precisa de ter desenvolvido para que a mesma passe no teste. Adicionalmente, a caracterização apresentada na Secção 3 dispõe de uma visão mais global sobre a complexidade dos testes dos grupos, as versões dos testes e os built-ins que aparecem nos testes. Através do uso da aplicação desenvolvida ou do MongoDB, implementações parciais da linguagem podem agora filtrar os testes pelas funcionalidades que a implementação já possui e testar o seu trabalho nas mesmas.

Bibliografia

[1] "especificação da linguagem ecmascript® , 6.0 edition / june 2015," acedido a 2020-12-21. [Online]. Available: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf

[2] "Spidermonkey is mozilla's javascript engine written in c and c++," acedido a 2020-12-21. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[3] "V8 - google's open source high-performance javascript and webassembly engine, written in c++." acedido a 2020-06-07. [Online]. Available: https://v8.dev/

[4] "Chakra interpretador js utilizado pelas aplicações windows," acedido a 2020-12-21. [Online]. Available: https://github.com/microsoft/ChakraCore

[5] "Ambiente de programação multi-tier para o javascript," acedido a 2020-12-21. [Online]. Available: https://github.com/manuel-serrano/hop

[6] "Um motor javascript que corre em cima do motor javascript do browser," acedido a 2020-12-21. [Online]. Available: https://github.com/mbbill/JSC.js

[7] "Compilador ahead-of-time e um runtime para o ecmascript," acedido a 2020-12-21. [Online]. Available: https://github.com/toshok/echojs

[8] "Test262 - official ecmascript conformance test suite," acedido a 2020-06-07. [Online]. Available: https://github.com/tc39/test262/

[9] "Base de dados nosql," acedido a 2020-12-21. [Online]. Available: https://www.mongodb.com/

[10] P. T. Simon Holm Jensen, Anders Møller, "Type analysis for javascript," *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, 2009.

[11] P. G. Christopher Anderson, Sophia Drossopoutou, "Towards type inference for javascript," *ECOOP'05: Proceedings of the 19th European conference on Object-Oriented Programming*, 2005.

[12] M. T. Gavin Bierman, Martím Abadi, “Understanding typescript,” *ECOOP 2014 – Object-Oriented Programming*, 2014.

[13] S.-y. G. Brian Hackett, “Fast and precise hybrid type inference for javascript,” *PLDI ’12: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[14] R. J. Ravi Chugh, David Herman, “Dependent types for javascript,” *ACM SIGPLAN Notices, Volume 47, Issue 10*, 2012.

[15] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris, “Safe & efficient gradual typing for typescript,” *ACM SIGPLAN Notices, Volume 50, Issue 1*, 2015.

[16] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, “Type inference for static compilation of javascript,” *ACM SIGPLAN Notices, Volume 51, Issue 10*, 2016.

[17] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, “Safe: Formal specification and implementation of a scalable analysis framework for ecmascript,” 2012.

[18] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, “Jsai: a static analysis platform for javascript,” *FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[19] S. R. Changhee Park, “Scalable and precise static analysis of javascript applications via loop-sensitivity,” *Dagstuhl Artifacts Series*, 2015.

[20] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey, and C. Zhang, *Combining String Abstract Domains for JavaScript Analysis: An Evaluation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017.

[21] K.-M. C. Dongseok Jang, “Points-to analysis for javascript,” *Proceedings of the ACM Symposium on Applied Computing*, 2009.

[22] J. Fragoso Santos, P. Maksimović, D. Naudziuniene, T. Wood, and P. Gardner, “Javert: Javascript verification toolchain,” *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 1–33, 12 2017.

[23] J. F. Santos, P. Maksimović, G. Sampaio, and P. Gardner, “Javert 2.0: compositional symbolic execution for javascript,” *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–31, 01 2019.

[24] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, vol. 49, pp. 87–100, 01 2014.

[25] D. Park, A. Ştefănescu, and G. Roşu, "Kjs: A complete formal semantics of javascript," *ACM SIGPLAN Notices*, vol. 50, pp. 346–356, 06 2015.

[26] A. Charguéraud, A. Schmitt, and T. Wood, "Jsexplain: A double debugger for javascript," *WWW '18: Companion Proceedings of the The Web Conference 2018*, pp. 691–699, 04 2018.

[27] R. B. F. Manuel Serrano, "Dynamic property caches: a step towards faster javascript proxy objects," *CC 2020: Proceedings of the 29th International Conference on Compiler Construction*, 2020.

[28] P. Thiemann, "Towards a type system for analyzing javascript programs," *Lecture Notes in Computer Science*, vol. 3444, 04 2005.

[29] S. Maffeis, J. Mitchell, and A. Taly, "An operational semantics for javascript," in *Programming Languages and Systems*, 12 2008, pp. 307–325.

[30] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," *ECOOP, Lec- Ture Notes in Computer Science*, pp. 126–150, 06 2010.

[31] "Racket - general-purpose programming language," acedido a 2020-06-07. [Online]. Available: https://racket-lang.org/

[32] P. Gardner, S. Maffeis, and G. Smith, "Towards a program logic for javascript," *Sigplan Notices - SIGPLAN*, vol. 47, pp. 31–44, 01 2012.

[33] J. Politz, M. Carroll, B. Lerner, J. Pombrio, and S. Krishnamurthi, "A tested semantics for getters, setters, and eval in javascript," *ACM SIGPLAN Notices*, vol. 48, pp. 1–16, 10 2012.

[34] "Coq - interactive formal proof management system," acedido a 2020-06-07. [Online]. Available: https://coq.inria.fr/

[35] "Ocaml - general-purpose, multi-paradigm programming language," acedido a 2020-06-07. [Online]. Available: https://ocaml.org/

[36] "K - rewrite-based executable semantic framework," acedido a 2020-06-07. [Online]. Available: http://www.kframework.org/index.php/Main_Page

[37] A. ŞtefănescuŞtefan CiobâcăRadu MereutaBrandon M. MooreTraian Florin ŞerbănuţăGrigore Roşu, "All-path reachability logic," *Logical Methods in Computer Science*, 2019.

[38] P. Gardner, G. Smith, C. Watt, and T. Wood, “A trusted mechanised specification of javascript: One year on,” *SIGPLAN Not.*, vol. 9206, pp. 3–10, 07 2015.

[39] P. M. e. P. G. Gabriela Sampaio, Fragoso Santos, “A trusted infrastructure for symbolic analysis of event-driven web applications,” *European Conference on Object-Oriented Programming (ECOOP 2020)*, vol. 166, pp. 1–28, 2020.

[40] ——, “A trusted infrastructure for symbolic analysis of event-driven web applications,” *European Conference on Object-Oriented Programming (ECOOP 2020)*, vol. 166, 2020.