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

Miguel Maria Marçalo Pires Trigo
ist188072
miguel.trigo@tecnico.ulisboa.pt

Instituto Superior Técnico
Av. Rovisco Pais, 1
1049-001 Lisboa
Tel: +351 218 417 000
mail@tecnico.ulisboa.pt

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 escrito em inglês. Uma implementação de referência do standard ECMAScript enfrenta vários desafios no processo de teste. O nosso projeto de 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 vai utilizar 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 fornecerá um mecanismo de filtragem dos testes por feature e um processo de automação do mecanismo de testes. O projeto tese também inclui o desenvolvimento de uma implementação de referência das secções core do standard ECMAScript 6. Esta implementação vai ser testada utilizando a infra-estrutura desenvolvida no projeto tese.

Keywords: ECMAScript · Specification Language · Reference Interpreters · Dynamic Languages · Test262 · OCaml

Conteúdo

1 Introdução

JavaScript (JS) é a linguagem de programação standard 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 [6], que consiste num documento complexo escrito em inglês. JavaScript é uma linguagem complicada, que 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.

Como a maioria das linguagens dinâmicas, a linguagem JavaScript é normalmente interpretada. De entre os interpretadores mais relevantes destacam-se: o spidermonkey [11], o V8 [14] e o chakra [3]. 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 [1], o JSC [13] e o echoJS [4]. Devido ao carácter dinâmico da linguagem, os compiladores puramente estáticos são menos eficientes que os interpretadores baseados em just-in-time compilation.

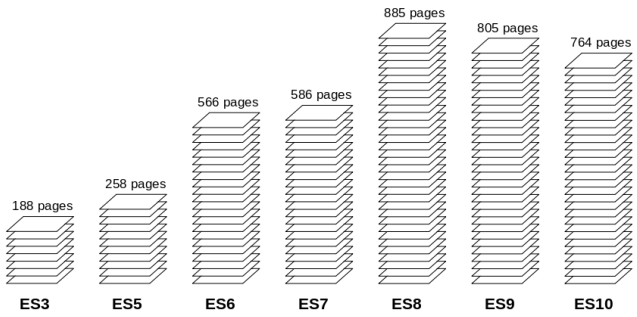


Figura 1. Evolução do número de páginas do documento oficial do standard ECMAScript

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

[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

estes *corner cases*. Assim, durante o desenvolvimento 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 corner cases da linguagem?*
   Mesmo a Test262 [12], 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 também agora 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 temos como objetivo construir uma infra-estrutura que auxilie o desenvolvimento de implementações de referência da linguagem JavaScript. A infra-estrutura a ser 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 vai guardar a localização dos testes e os metadados, informação adicional, associados aos mesmos. O cálculo dos metadados é uma contribuição essencial desta tese, porque até ao momento os testes da Test262 [12] não têm a si associados metadados formais; os poucos metadados estão inseridos

em comentário nos próprios testes. Estes metadados vão mais tarde permitir 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 vai servir para armazenar os resultados dos testes executados e comparar os resultados da versão corrente com os resultados das versões anteriores. Esta base de dados vai-nos permitir 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.

Prevemos que esta infra-estrutura venha facilitar significativamente o desenvolvimento de implementações de referência para linguagem JavaScript.

Para avaliar a nossa infraestrutura vamos criar uma implementação de referência do Core da versão 6 da linguagem JavaScript [6]; mais concretamente, vamos implementar as secções 7, 9, 12 e 13 do standard utilizando a infra-estrutura proposta para gestão do desenvolvimento. Adicionalmente, esta infra-estrutura vai ser também utilizada por outros dois alunos de mestrado que vão implementar as restantes secções do standard.

A base de dados de testes vai conter os metadados associados aos testes da Test262 [12]. Atualmente estes testes já incluem alguns metadados em comentários embebidos no código dos próprios testes, nomeadamente: versão da linguagem, strict vs non-strict (strict avalia os argumentos, enquanto que non-strict não precisa de fazer esta verificação) e secção do standard. Uma parte essencial desta tese será definir um formato canónico para representação dos metadados dos testes, que irá abranger os metadados já existentes, bem como metadados adicionais. A definição dos metadados adicionais a incluir vai ser informada pela nossa experiência de implementação do standard: à medida que tivermos necessidade de novos metadados, iremos calculá-los e adicioná-los à base de dados. De entre os metadados que consideramos adicionar destacam-se: as construções sintáticas que ocorrem nos testes (número de ifs, whiles,..), os algoritmos internos do standard exercitados pelo teste e as bibliotecas built-in utilizadas pelos testes.

As bases de dados acima referidas vão ser desenvolvidas em mongoDB [2]. 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.

O relatório estrutura-se como se descreve em seguida. Na Secção 2 resumimos os objectivos deste projeto de tese. A Secção 3 consiste num *background* do ECMAScript versão 6, com foco principal nas secções 7, 9, 12 e 13, que constituem o core da linguagem. Na Secção 4 apresentamos uma overview da linguagem ECMA-SL, que vai ser utilizada para implementação do nosso interpretador de referência. Na Secção 5 falamos dos trabalhos relacionados com o tema deste projeto, enquanto que na Secção 6 descrevemos a metodologia e o design utilizado na implementação do projeto apresentado. A Secção 7, por sua vez, consiste na descrição da metodologia de avaliação a utilizar durante a execução da proposta. Por fim, a Secção 8 termina o relatório, apresentando as principais conclusões e um sumário da proposta.

2 Objetivos

O objetivo principal desde projeto é o desenho e implementação de uma infra-estrutura para gerir o processo de desenvolvimento de implementações de referência do standard ECMAScript. Para tal dividimos o objetivo principal do projeto em três sub-objetivos.

1. Caracterização da testsuite Test262 [12];
2. Infra-estrutura para testagem de implementações de referência do standard ECMAScript;
3. Implementação do core da versão 6 do standard ECMAScript utilizando a infra-estrutura concebida;

Caracterização da testsuite Test262 O primeiro objetivo do projeto vai ser analizar e caracterizar os testes presentes na bateria de testes oficial do JavaScript, a Test262. Como parte deste objectivo vamos definir um formato canónico para representação dos metadados associados aos testes e calcular para cada teste os seus metadados. Para além dos metadados já existentes nos testes, o formato canónico a definir incluirá metadados adicionais, como sejam: o número de linhas, número de ifs/loops/etc, número de variáveis, entre outros que podem ser acrescentados facilmente mais tarde. Uma ferramenta que pensamos usar para extração de informação adicional sobre os testes é o jalangi [7], uma das ferramentas mais utilizadas para análise dinâmica de código JavaScript.

Infra-estrutura para testagem de implementações de referência do standard ECMAScript Este objetivo visa a criação da infra-estrutura para suporte ao desenvolvimento de implementação de referência do Standard ECMAScript. Esta infra-estrutura vai facilitar o processo de testagem de implementações de referência do standard ECMAScript. Para facilitar este processo, a infra-estrutura

a desenvolver vai permitir ao utilizador a filtragem de testes por feature, permitindo a obtenção dos testes relevantes para as características a serem testadas, bem como a execução automática dos testes filtrados e armazenamento dos respectivos resultados. Os resultados dos testes vão ser guardados na segunda base de dados pelo que também será necessário o desenvolvimento de mecanismos para facilitar a interação com esta base de dados.

Implementação do core da versão 6 do standard ECMAScript utilizando a infra-estrutura concebida O último objetivo a alcançar neste projeto é a criação de uma implementação do standard ECMAScript para testarmos a infra-estrutura desenvolvida. Nós escolhemos implementar o core da versão 6 do standard, descrito nas secções 7 (operações abstradas), 9 (comportamento de objetos exóticos e comuns), 12 (linguagem ECMAScript: expressões) e 13 (linguagem ECMAScript: afirmações e declarações). As restantes secções do Standard prendem-se sobretudo com a descrição do comportamento de funções e classes (secção 14), scripts e módulos (secção 15), erros e extensões da linguagem (secção 16) e bibliotecas built-in (secções 17 a 25)

3 ECMAScript Standard

ECMAScript Standard é o documento oficial que define a linguagem JavaScript. Este documento é regularmente atualizado pelo comité TC39, responsável pela manutenção do standard. O standard encontra-se atualmente na sua décima versão. Contudo, neste projeto, vamos ocupar-nos da versão 6, daqui em diante referida como ES6, na qual foram introduzidas a maioria das features atuais da linguagem. Acreditamos que os resultados deste trabalho serão extensíveis para as versões mais recentes.

O standard define os tipos, valores, objetos, propriedades, funções, e a sintaxe e semânticas do JavaScript que devem existir em qualquer implementação da linguagem. Notar que o standard permite que as implementações da linguagem forneçam tipos, valores, objetos, propriedades e funções adicionais. As subsecções seguintes apresentam uma descrição detalhada do core da linguagem ES6 que será implementado neste trabalho.

3.1 Visão Global do ECMAScript

A semântica da linguagem ES6 pode dividir-se em três componentes principais: expressões e comandos, bibliotecas built-in e funções internas. A estrutura da semântica da linguagem ES6 pode ser observada na figura 2, onde também podem ser oservados os vários elementos presentes nas bibliotecas *built-in*.

A componente de *expressões e comandos* descreve os comportamentos das construções sintáticas do ES6, detalhando a semântica das diversas expressões (expressões de atribuição, operatores built-in, ...), comandos (comandos de loops, comandos condicionais, ...) e tipos built-in (Undefined, Null, Boolean, Number, String, and Object).

As funções internas da linguagem são utilizadas para definir a semântica tanto das expressões e comandos como das bibliotecas built-in. Estas funções não estam expostas para além do âmbito da linguagem interna, isto é, um programa ES6 não utiliza nenhuma destas funções diretamente.

Finalmente, as *bibliotecas built-in* contêm todos os objetos internos disponíveis quando um programa ES6 é executado. Estes objetos expõem inúmeras funções implementadas pela propria linguagem, incluindo funções para manipulação de números, texto, arrays, objetos, entre outros. Uma explicação mais detalhada vai ser dada nas secções seguintes.

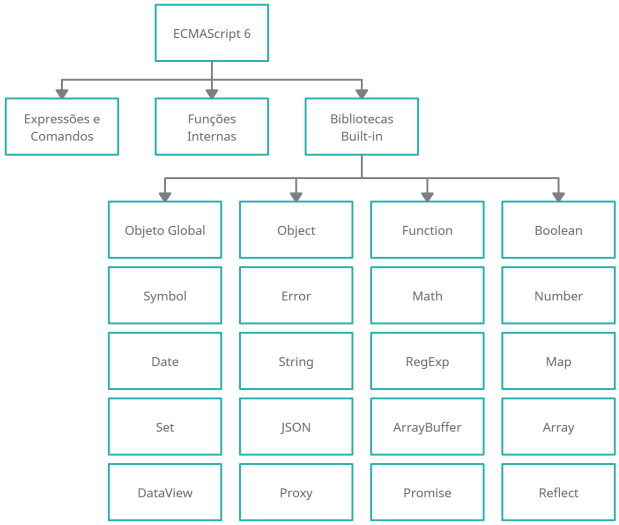


Figura 2. Estrutura do ECMAScript 6

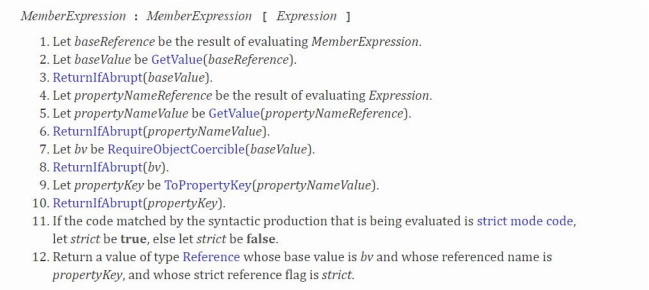
3.2 Expressões e Comandos

O objetivo desta sub-secção é apresentar como o standard ECMAScript 6 define os comandos e as expressões da linguagem JavaScript. De seguida, vão ser apresentados dois exemplos, um para as expressões e outro para os comandos. Estes exemplos são acompanhados de uma descrição dos mesmos juntamente com o snippet da sua implementação no standard. Para o caso das expressões vai ser apresentada a expressão `MemberExpression`, enquanto que para os comandos vai ser usado o comando `if`.

A Figura 3 mostra o snippet do standard que descreve a semântica do acesso a uma propriedade de um objeto. Para avaliar a expressão:

```
MemberExpression[Expression]
```

a linguagem começa por avaliar a expressão `MemberExpression`, obtendo uma referência para o seu valor que é atribuida à variável `baseReference` (passo 1). De seguida, a linguagem obtem o valor da referência e atribui-o à variável `baseValue` (passo 2). Para tal, a linguagem faz uso da função interna `GetValue`, cujo objetivo é determinar o valor de uma referência. Se a função `GetValue` gerar um erro, este é retornado imediatamente (passo 3). Senão, a linguagem avalia a expressão `Expression` (passo 4) e atribui o valor da referência obtida à variável `propertyNameValue` (passo 5), podendo retornar imediatamente um erro (passo 6). Depois de obter os valores das expressões `MemberExpression` e `Expression`, respetivamente `baseValue` e `propertyNameValue`, a linguagem invoca a função `RequireObjectCoercible` sobre o valor `baseValue` para transformar este valor num objeto caso este não o seja (passo 7). Analogamente, a linguagem invoca a função `ToPropertyKey` sobre o valor `propertyNameValue` para transformar este valor numa propriedade caso este não o seja (passo 9). Se quer o passo 7 como o passo 9 retornarem erros, a avaliação termina imediatamente com erro (passos 8 e 10, respetivamente). Finalmente, a linguagem combina o objeto e a propriedade obtidos nos passos 7 e 9 para construir uma nova referência, correspondente ao resultado da avaliação (passo 12). A referência retornada guarda também informação sobre o contexto onde foi criada, nomeadamente se se trata de código strict ou non-strict (passo 11).



MemberExpression : *MemberExpression* [*Expression*]

1. Let *baseReference* be the result of evaluating *MemberExpression*.
2. Let *baseValue* be GetValue(*baseReference*).
3. ReturnIfAbrupt(*baseValue*).
4. Let *propertyNameReference* be the result of evaluating *Expression*.
5. Let *propertyNameValue* be GetValue(*propertyNameReference*).
6. ReturnIfAbrupt(*propertyNameValue*).
7. Let *bv* be RequireObjectCoercible(*baseValue*).
8. ReturnIfAbrupt(*bv*).
9. Let *propertyKey* be ToPropertyKey(*propertyNameValue*).
10. ReturnIfAbrupt(*propertyKey*).
11. If the code matched by the syntactic production that is being evaluated is strict mode code, let *strict* be **true**, else let *strict* be **false**.
12. Return a value of type Reference whose base value is *bv* and whose referenced name is *propertyKey*, and whose strict reference flag is *strict*.

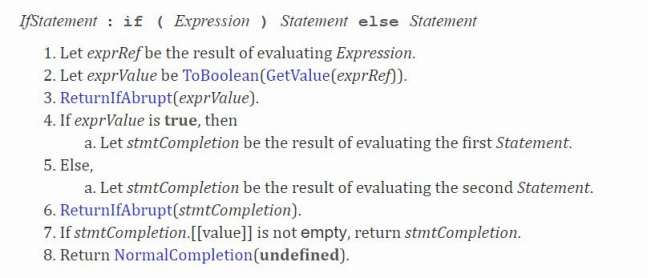
Figura 3. ECMAScript da expressão `MemberExpression`

A Figura 4 mostra o snippet do standard que descreve a semântica do comando if. Para avaliar o comando:

```
if (Expression) Statement1 else Statement2.
```

a linguagem começa por avaliar a `Expression` associando a referência obtida ao valor da variável `exprRef` (passo 1). A referência obtida no passo anterior deve denotar um booleano. Para garantir que tal aconteça, a linguagem vai transfomar o valor da referência em booleano, usando para tal as funções `GetValue` e `ToBoolean`, e guardando o resultado da sua aplicação sucessiva na variável `exprValue` (passo 2). Caso o passo anterior gere um erro, a linguagem vai retornar de imediato o erro (passo 3). De seguida, a linguagem procede de maneira

diversa dependendo do valor da variável `exprValue`. Em ambas os casos apenas se vai mudar o valor associado à variável `stmtCompletion`. Caso a variável `exprValue` tenha valor `true`, a variável `stmtCompletion` vai guardar o resultado da avaliação do primeiro `Statement` (passo 4). Caso contrário, a variável `stmtCompletion` vai guardar o resultado da avaliação do segundo `Statement` (passo 5). Em seguida, é verificado se a variável `stmtCompletion` contém um erro; se for o caso, o erro é retornado de imediato (passo 6). Por fim, a linguagem retorna a completion obtida caso o seu atributo value não seja vazio; caso contrário, é retornada uma nova completion com atributo value `undefined`.



IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ToBoolean(GetValue(*exprRef*)).
3. ReturnIfAbrupt(*exprValue*).
4. If *exprValue* is **true**, then
    a. Let *stmtCompletion* be the result of evaluating the first *Statement*.
5. Else,
    a. Let *stmtCompletion* be the result of evaluating the second *Statement*.
6. ReturnIfAbrupt(*stmtCompletion*).
7. If *stmtCompletion*.[[value]] is not empty, return *stmtCompletion*.
8. Return NormalCompletion(**undefined**).

Figura 4. ECMAScript do comando If

3.3 Funções Internas

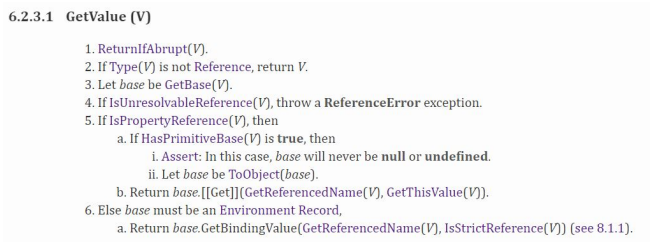
As funções internas têm o objetivo de auxiliar a descrição da semântica da linguagem. O standard contém a definição de todas as funções internas com os respectivos algoritmos. As funções internas presentes na linguagem vão desde a conversão de tipos, funções de teste, métodos internos dos objetos, funções para operar sobre *Property Descriptors* e funções para operar ou aceder a componentes de referências.

Muitas funções internas estão interligadas, ou seja a definição de uma função interna vai invocar outras funções internas. Isto faz com que a compreensão do comportamento das funções internas possa ser difícil.

De seguida, vai ser apresentada a descrição mais detalhada do comportamento de uma função interna do Standard do ECMAScript, a função `GetValue`, bem como um snippet da sua definição como apresentado no standard.

A função interna `GetValue` vai receber como argumento a referência `V` cujo valor se pretende calcular. Em JavaScript uma referência pode ser entendida como um apontador para uma propriedade de um objeto. Assim, uma referência em JavaScript é composta por dois elementos principais: o endereço do objeto, designado por base da referência, e o nome da propriedade.

Para determinar o valor denotado por uma referência, a linguagem começa por verificar se existe um erro no argumento, servindo-se da função interna



6.2.3.1 GetValue (V)

1. ReturnIfAbrupt(*V*).
2. If Type(*V*) is not Reference, return *V*.
3. Let *base* be GetBase(*V*).
4. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
5. If IsPropertyReference(*V*), then
    a. If HasPrimitiveBase(*V*) is **true**, then
        i. Assert: In this case, *base* will never be **null** or **undefined**.
        ii. Let *base* be ToObject(*base*).
    b. Return *base*.[[Get]](GetReferencedName(*V*), GetThisValue(*V*)).
6. Else *base* must be an Environment Record,
    a. Return *base*.GetBindingValue(GetReferencedName(*V*), IsStrictReference(*V*)) (see 8.1.1).

Figura 5. ECMAScript da função interna GetValue

`ReturnIfAbrupt` e retornando de imediato o erro obtido (passo 1). De seguida, caso o tipo do argumento não seja `Reference`, o argumento é igualmente retornado de imediato (passo 2). Depois, a linguagem obtém a base da referência inicial, através da função interna `GetBase`, atribuindo o valor obtido à variável `base` (passo 3).

De seguida, a linguagem verifica se `V` se trata de uma referência não resolúvel (`Unresolvable Reference`), retornando de imediato um `ReferenceError` caso esta o seja (passo 4). Uma referência diz-se não resolúvel se tiver por base o valor `undefined`.

Por fim, a linguagem vai determinar o tipo da referência e, conforme o tipo, proceder de maneira diferente. Existem dois tipos de referências: `PropertyReferences`, que denotam propriedades de objetos, e `Environment Record References`, que denotam variáveis do programa em execução.

Caso a referência `V` se trate de uma `property reference` (passo 5), a linguagem verifica se a base da referência é um tipo primitivo (passo 5.a), em cujo caso, esta é convertida para um objeto usando a função `ToObject` (passo 5.a.ii). Finalmente, a linguagem retorna o resultado da execução da função interna `get` sobre a base da referência, passando como parâmetros o nome da propriedade.

Caso a referência se trate de uma `Environment Record Reference` a linguagem invoca o método `GetBindingValue` da base com o parâmetro do nome da propriedade (passo 6.a).

3.4 Bibliotecas built-in

As bibliotecas built-in contêm diversos tipos de objetos, os quais são sempre acessíveis em qualquer script ou módulo JavaScript. Estes objetos incluem o objeto global, além de outros objetos fundamentais como *Object*, *Function*, *Boolean*, *Symbol*, vários objectos *Error*, objectos de manipulação numérica como *Math*, *Number* e *Date*, objectos de processamento de texto, *String* e *RegExp*, objectos que representam coleções indexadas, de onde se destaca o objeto *Array*, coleções mapeadas como o *Map* e o *Set*, objectos de suporte a estruturas de dados, como os objectos *JSON*, *ArrayBuffer* e *DataView*, objectos de suporte a controlo de abstrações incluindo funções do gerador e objectos *Promise*, e objectos de refleção incluindo *Proxy* e *Reflect*.

4 ECMA-SL

A ECMA-SL é uma linguagem de programação intermédia para especificação e análise do standard JavaScript. Como parte deste projeto vamos utilizar a infra-estrutura de testes estabelecida para suportar o desenvolvimento de um novo interpretador de referência para ECMAScript 6 em ECMA-SL. O novo interpretador de referência será desenvolvido tendo por base um interpretador de referência para ECMAScript 5 também desenvolvido em ECMA-SL no contexto de uma tese de mestrado anterior.

Nesta secção vamos apresentar a linguagem ECMA-SL, explicando como esta pode ser utilizada para implementar as funções internas e construções do JavaScript. Vamos recorrer aos exemplos introduzidos na Secção 3 em forma de pseudo-código, mostrando como os mesmos podem ser implementados em ECMA-SL.

4.1 Expressões e Comandos

A expressão `MemberExpression` implementada ECMA-SL está apresentada na Figura 6, onde se pode observar a semelhança com a sua representação no standard ES6. Inicializando com o uso da função auxiliar `JS_Interpreter_Expr` para avaliar a `MemberExpression` e a `Expression`, seguido da obtenção do valor de ambas pela função interna `GetValue`. Caso o valor da avaliação de alguma das expressões seja inválido, o programa vai retornar de imediato, através da macro `ReturnIfAbrupt`. Depois, é verificado se o valor da avaliação da `MemberExpression` pode ser transformado num objeto, recorrendo a função auxiliar `RequireObjectCoercible`, e transforma o valor da avaliação da `Expression` numa `Property Key`. Se algum dos dois passos anteriores gerar um valor inválido volta a ser chamada a macro `ReturnIfAbrupt`. A seguir, é inicializada uma variável `strict` que será um booleano, `true` caso o código a ser corrido esteja em modo *strict* e `false` caso contrário. Por fim, é criada um objeto de tipo `Reference` com o valor igual ao valor da avaliação da `MemberExpression`, o nome da referência igual ao valor da avaliação da `Expression` e com a flag *strict* igual ao valor da variável `strict`.

A representação em ECMA-SL do comando `if` vai ser muito próxima da definição presente no standard do JavaScript, como pode ser observado na Figura 7. Fazendo uso de funções auxiliares, como é o caso da função `JS_Interpreter_Expr` que vai retornar a avaliação da expressão dada como argumento. A função auxiliar `JS_Interpreter_Expr` vai ser usada no primeiro passo avaliando o valor da `Expression` e guardando o valor na variável `exprRef`. O valor desta variável, obtido através da função interna `GetValue`, vai ser convertido em booleano e armazenado na variável `exprValue`. Caso a variável `exprValue` apresente algum erro, este mesmo erro vai retornar de imediato. Caso contrário, isto é se `exprValue` não tiver nenhum erro, é verificado o valor do mesmo. Se este é `true`, chamamos recursivamente a função `JS_Interpreter_Stmt` no `Statement1` e atribuímos o valor obtido à variável `stmtCompletion`. Se este é `false`, procedemos de forma equivalente com o `Statement2`. Por fim, se o `stmtCompletion`

```
| { type: "MemberExpression", object: MemberExpression, property: Expression } ->
{
    base := JS_Interpreter_Expr(MemberExpression, scope);
    baseValue := GetValue(baseReference);
    ReturnIfAbrupt(baseValue);
    propertyNameReference := JS_Interpreter_Expr(Expression, scope);
    propertyNameValue := GetValue(propertyNameReference);
    ReturnIfAbrupt(propertyNameValue);
    bv := RequireObjectCoercible(baseValue);
    ReturnIfAbrupt(bv);
    propertyKey := ToPropertyKey(propertyNameValue);
    ReturnIfAbrupt(propertyKey);
    if (isContainedInStrictCode(scope)) {
        strict := true
    } else {
        strict := false
    };
    return newPropertyReference(baseValue, propertyKey, strict)
}
```

Figura 6. Código ESL da expressão MemberExpression

obtido não corresponder a um erro e se o seu valor for diferente de `empty`, então este é simplesmente retornado.

```
| { type: "IfStatement", test: Expression, consequent: Statement1, alternate: Statement2 } ->
{
    exprRef := JS_Interpreter_Expr(Expression, scope);
    exprValue := ToBoolean(GetValue(exprRef));
    ReturnIfAbrupt(exprValue);
    if (exprValue = true) {
        stmtCompletion := JS_Interpreter_Stmt(Statement1, scope)
    }
    else {
        stmtCompletion :=JS_Interpreter_Stmt(Statement2, scope)
    }
    ReturnIfAbrupt(stmtCompletion);
    if (!(stmtCompletion.value = 'empty)) {
        return stmtCompletion
    };
    return NormalCompletion('undefined)
}
```

Figura 7. Código ESL da expressão If

4.2 Funções Internas

A Figura 8 apresenta a função `GetValue` escrita na linguagem ECMA-SL. Esta função utiliza diversas funções auxiliares que devem ser implementadas em paralelo para que a execução da função `GetValue` corra sem problemas. O código em ECMA-SL é semelhante à definição presente no standard, onde a função `GetValue` recebe um argumento que vai ser a referência a ser analizada. Nos primeiros passos começa por verificar se a referência recebida contém um erro, servindo-se da macro auxiliar `ReturnIfAbrupt`, ou se não é de facto do tipo `Reference` através da expressão `typeof` para verificar o tipo do argumento. Caso alguma das condições anteriores se verifique, a função retorna de imediato. De seguida, a base da referência é associada à variável `base`. Depois, tal como no standard, verificamos se a referência a ser analizada é uma

referência não resolúvel, retornando uma exceção nesse caso. Se a referência for resolúvel, temos de verificar se se trata de uma `Property Reference` ou de um `Environment Record Reference`. Caso se trate de uma `Property Reference` e a base da referência seja primitiva e diferente de `null` e `undefined`, a variável `base` vai ser convertida num objeto, através da função `ToObject`. Finalmente, invocamos a função interna `get` sobre a base da referência. Caso a referência seja um `Environment Record Reference`, vai retornar a chamada da função `GetBindingValue` sobre a base da referência anteriormente obtida.

```
function GetValue(V) {
    ReturnIfAbrupt(V);
    if (!((typeof V) = Reference)){
        return V
    };
    base := GetBase(V);
    if (IsUnresolvableReference(V) = true){
        throw ReferenceErrorException()
    };
    if (IsPropertyReference(V) = true){
        if (HasPrimitiveBase(V) = true){
            Assert !(base = null) &&& !(base = undefined);
            base := ToObject(base)
        };
        return {base.get}(GetReferencedName(V), GetThisValue(V))
    };

    else{
        return GetBindingValue(base, GetReferencedName(V), IsStrictReference(V))
    }
}
```

Figura 8. Código ESL da função GetValue

5 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 ([39], [20], [27], [17], [35], [15], [38]), interpretadores abstratos ([29], [41], [18], [36]), análises *points-to* ([21]), lógicas de Hoare ([22], [30]), semântica operacional ([16], [33], [19]), representações intermédias e compiladores ([32]). Em seguida, apresentamos os trabalhos relacionados mais relevantes para o projeto tese a ser desenvolvido, organizando-os numa linha temporal dividida nas Tabelas 1 e 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 [40]. 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) [31]. Esta semântica foi usada para analisar várias propriedades de segurança de aplicações web e mashups.
2010	Guha et al. [28] definem a linguagem λJS, um cálculo formal que captura as características mais fundamentais do ES3. λJS vem com um interpretador *Racket* [10] e um tradutor de programas ES3 em expressões λJS.
2012	Gardner et al. [25] 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. [34] 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* [16], uma formalização do standard ES5 escrito em *Coq* [5], e *JSRef* um interpretador executável de referências extraido do *Coq* para o *OCaml* [9], provado correto em relação ao *JSCert* e testado usando os testes da bateria Test262 [12].
2015	Park et al. apresenta *KJS* [33], a semântica formal do ES5 mais completa até à data. O *KJS* foi desenvolvido usando a framework *K* [8], 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. [26] estendem o projecto *JSCert* para suportar *arrays* ES5. Para tal, os autores ligam o *JSRef* à implementação da biblioteca Array do motor V8 [14] 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.

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

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* [19], 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. [23] 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 [24] a primeira ferramenta que sintetiza automaticamente o analizador e que traduz AST-IR diretamente de uma especificação da linguagem dada.

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

6 Desenho e Metodologia

6.1 Infra-estrutura de Testes

Os testes presentes na bateria oficial de testes do JavaScript, a Test262 [12], são compostos 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 9. 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 Figura 9 mostra um snippet de um teste da bateria de testes oficial do JavaScript, a Test262, para a função `for`. Como pode ser verificado o teste contém

nas duas primeiras linhas a secção de direitos de autor, seguida da secção de *Frontmatter* entre as linhas 4 e 13. Na secção de *Frontmatter* do teste apresentado podem observar-se as palavras-chaves *info*, *es5id*, *description* e *negative*. A palavra-chave *info* contém uma citação direta da secção a ser testada do standard, a palavra-chave *es5id* contém a secção da versão 5.1 do standard a ser testada. Os dados associados à *description* vão conter uma breve descrição do teste a ser executado. Por fim, os dados da secção *negative* vão explicitar a fase e o erro que devem ser obtidos ao correr o teste, nas subsecções *phase* e *type* respetivamente. A secção *body* contém o código a ser testado.

```
1  // Copyright 2009 the Sputnik authors.  All rights reserved.
2  // This code is governed by the BSD license found in the LICENSE file.
3
4  /*---
5  info: Blocks within "for" braces are not allowed
6  es5id: 12.6.3_A8_T2
7  description: >
8      Checking if execution of "for(index=0; {index++;index<100;};
9      index*2;) {  arr.add(""+index);}" fails
10 negative:
11   phase: parse
12   type: SyntaxError
13 ---*/
14
15 $DONOTEVALUATE();
16
17 var arr = [];
18
19 ////////////////////////////////////////////////////////////////////////////////
20 //CHECK#1
21 for(index=0; {index++;index<100;}; index*2;) {  arr.add(""+index);};
22 //
23 ////////////////////////////////////////////////////////////////////////////////
```

Figura 9. Snippet de um teste da expressão `for`

Frontmatter A secção de `Frontmatter` faz uso de palavras-chave para organizar os metadados do teste. Algumas das palavras-chave mais relevantes para os metadados a serem considerados no nosso projeto tese vão ser:

1. *esid*: é utilizada sempre que uma nova feature é testada, contendo o identificador hash da porção do ECMAScript associado à nova feature.
2. *negative*: quando esta palavra-chave está presente significa que o teste espera lançar um erro de um tipo específico. A informação associada é um dicionário com um tipo que vai ser a string correspondente ao nome do erro a ser retornado e a fase na qual se espera o retorno do erro.

3. *flags*: esta palavra-chave tem associado uma lista de booleanos para cada propriedade. Estas propriedades são *onlyStrict* (o teste apenas é executado em modo *strict*), *noStrict* (o teste apenas é executado em modo "*sloppy*"), *module* (o texto fonte deve ser interpretado como código modulo), *raw* (executa o teste sem nenhuma modificação, o que implica que corre como *noStrict*), *async* (apenas para testes assíncronos), *generated* (flag que denota os ficheiros criados pelo teste), *CanBlockIsFalse* e *CanBlockIsTrue* (apenas executa testes que tenham a propriedade *CanBlock* a *false* ou *true*, respetivamente).
4. *features*: esta palavra-chave vai conter uma lista com as features que vão ser usadas no teste.
5. *es5id*: esta palavra-chave indica o número da secção do standard ECMAScript 5.1 ou 3 que está a ser testada. Infelizmente, não há maneira de identificar se este valor se refere à versão 3 ou à versão 5.1.
6. *es6id*: esta palavra-chave indica o número da secção do standard ECMAScript 6 que está a ser testada.

É importante notar que nem todos os testes vêm adequadamente anotados com todos os metadados que lhes corresponderiam. Por exemplo é comum um teste não conter a palavra-chave *features*, ou então conter uma descrição muito incompleta das features usadas no teste. Analogamente, é usual encontrarem-se testes que, embora sejam suportados pela versão 5 do standard, não vêm anotados com a palavra-chave *es5id* e a respectiva secção. Uma parte importante deste projeto vai consistir no cálculo da informação completa referente aos metadados já incluídos da bateria de testes.

De forma a guardar os metadados dos testes da Test262, os valores associados às palavras-chave vão ser registados em ficheiros JSON juntamente com dados adicionais. A Figura 10 apresenta o ficheiro JSON para o teste da função `for` da bateria de testes oficial apresentado em cima. Devido à simplicidade do teste apresentado, o ficheiro JSON com os metadados apenas apresenta os dados associados à secção que está a ser testada, correspondendo aos dados da palavra-chave "*es5id*"e da palavra-chave "*negative*"e os seus sub-dados "*phase*"e "*type*", que indicam que o teste deve devolver um erro juntamente com a fase e o tipo do erro.

```
{
    "es5id": "12.6.3_A8_T2",
    "negative": {
        "phase": "parse",
        "type": "SyntaxError"
    }
}
```

Figura 10. json para o teste apresentado na Figura 9

7 Avaliação e Planeamento

O planeamento deste projeto tese baseia-se na divisão de tarefas de forma a alcançar os objetivos definidos na Secção 2 deste relatório. Para este efeito, foram definidas 4 tarefas principais, sendo que uma delas vai conter 4 sub-tarefas. Estas tarefas são apresentadas no diagrama de Gantt ilustrado na Figura 11 e passam a ser descritas em maior pormenor de seguida.

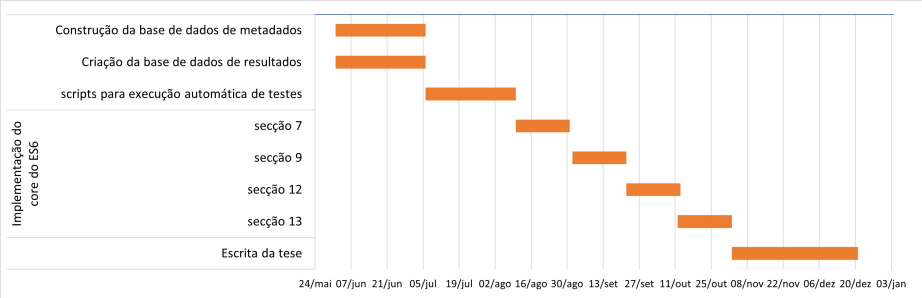


Figura 11. Diagrama de Gantt com o planeamento para o projeto tese

Construção da base de dados de metadados Nesta tarefa vai ser desenvolvida a base de dados que irá conter os metadados para cada teste. Esta base de dados vai permitir filtrar os testes desejados por feature e versão do standard.

Criação da base de dados de resultados Esta tarefa vai ser executada em paralelo com a tarefa anterior. A base de dados a desenvolver nesta tarefa vai guardar os resultados após a execução dos testes utilizando a implementação do standard a ser testada.

Scripts para execução automática de testes Esta tarefa vai basear-se na automatização da execução dos testes, bem como na filtragem dos testes pelas características desejadas usando a base de dados de metadados e armazenamento automático dos resutados obtidos na base de dados de resultados.

Implementação do core ES6 Esta tarefa vai ser sub-dividida em 4 tarefas mais pequenas. Vai ser composta por 4 sub-tarefas onde cada uma tem como objetivo a implementação em ECMA-SL da secção destinada. As secções a serem implementadas do ES6 vão ser as secções 7, 9, 12 e 13.

Escrita da Tese A última tarefa a executar vai consistir no reportar das execuções das tarefas anteriores e apresentação dos resultados e conclusões obtidas após a finalização do projeto tese.

7.1 Metodologia de Avaliação

A metodologia de avaliação a utilizar neste projeto tem dois elementos principais, respetivamente destinados à avaliação da infra-estrutura de testes desenvolvida e à avaliação da futura implementação de referência do core da versão 6 do standard. A implementação de referência a desenvolver vai ser testada utilizando o subconjunto de testes da Test262 apropriados. Este conjunto vai ser determinado utilizando a infra-estrutura de testes desenvolvida. Assim, a avaliação desta infra-estrutura vai fazer-se através da sua aplicação direta. Importa notar que esta infra-estrutura vai também ser utilizada por outros dois alunos cujas propostas de tese incluem a implementação de outras secções do standard.

8 Conclusão

Neste projeto tese propomos a realização de uma infra-estrutura de testes para implementações de referência do standard ECMAScript. Esta proposta vai permitir facilitar o processo de teste das implementações de referência do standard. A tese tem como principal objetivo a automatização dos testes e a filtragem dos mesmos por feature, o que vai permitir uma maior flexibilidade ao testar as implementações e uma maior eficiência no processo de teste de uma dada implementação. Ao simplificar o processo de teste das implementações de referência do standard, vamos conseguir descobrir erros nessas implementações mais rápido e mais precisamente.

Neste relatório, começamos com uma introdução ao standard ECMAScript, com maior foco nas secções correspondentes ao core da linguagem. Seguido de uma breve descrição da linguagem ECMA-SL, apresentando exemplos de código em ECMA-SL. Incluimos também uma descrição dos trabalhos científicos já realizados no âmbito da formalização da linguagem JavaScript, apresentando estudos realizados desde 2005 até 2020. Concluindo, com a apresentação do plano que pretendemos seguir durante a realização do projeto por forma a alcançar todos os objetivos definidos para o projeto e os métodos que vamos utilizar para avaliar o trabalho realizado.

Referências

1. Ambiente de programação multi-tier para o javascript, https://github.com/manuel-serrano/hop, acedido a 2020-12-21
2. Base de dados nosql, https://www.mongodb.com/, acedido a 2020-12-21
3. Chakra interpretador js utilizado pelas aplicações windows, https://github.com/microsoft/ChakraCore, acedido a 2020-12-21
4. Compilador ahead-of-time e um runtime para o ecmascript, https://github.com/toshok/echojs, acedido a 2020-12-21
5. Coq - interactive formal proof management system, https://coq.inria.fr/, acedido a 2020-06-07
6. especificação da linguagem ecmascript® , 6.0 edition / june 2015, http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf, acedido a 2020-12-21

7. Framework para escrita de análise dinâmica do javascript, https://github.com/Samsung/jalangi2, acedido a 2020-12-21
8. K - rewrite-based executable semantic framework, http://www.kframework.org/index.php/Main_Page, acedido a 2020-06-07
9. Ocaml - general-purpose, multi-paradigm programming language, https://ocaml.org/, acedido a 2020-06-07
10. Racket - general-purpose programming language, https://racket-lang.org/, acedido a 2020-06-07
11. Spidermonkey is mozilla's javascript engine written in c and c++, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, acedido a 2020-12-21
12. Test262 - official ecmascript conformance test suite, https://github.com/tc39/test262/, acedido a 2020-06-07
13. Um motor javascript que corre em cima do motor javascript do browser, https://github.com/mbbill/JSC.js, acedido a 2020-12-21
14. V8 - google's open source high-performance javascript and webassembly engine, written in c++., https://v8.dev/, acedido a 2020-06-07
15. Aseem Rastogi, Nikhil Swamy, C.F.G.B.P.V.: Safe efficient gradual typing for typescript (2015). https://doi.org/10.1145/2775051.2676971
16. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised javascript specification. vol. 49, pp. 87–100 (01 2014). https://doi.org/10.1145/2578855.2535876
17. Brian Hackett, S.y.G.: Fast and precise hybrid type inference for javascript (2012). https://doi.org/10.1145/2345156.2254094
18. Changhee Park, S.R.: Scalable and precise static analysis of javascript applications via loop-sensitivity (2015)
19. Charguéraud, A., Schmitt, A., Wood, T.: Jsexplain: A double debugger for javascript. pp. 691–699 (04 2018). https://doi.org/10.1145/3184558.3185969
20. Christopher Anderson, Sophia Drossopoutou, P.G.: Towards type inference for javascript (2005). https://doi.org/10.1007/11531142$_1$9
21. Dongseok Jang, K.M.C.: Points-to analysis for javascript (2009). https://doi.org/10.1145/1529282.1529711
22. Fragoso Santos, J., Maksimović, P., Naudziuniene, D., Wood, T., Gardner, P.: Javert: Javascript verification toolchain. Proceedings of the ACM on Programming Languages **2**, 1–33 (12 2017). https://doi.org/10.1145/3158138
23. Gabriela Sampaio, Fragoso Santos, P.M.e.P.G.: A trusted infrastructure for symbolic analysis of event-driven web applications. European Conference on Object-Oriented Programming (ECOOP 2020) **166**, 1–28 (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.28
24. Gabriela Sampaio, Fragoso Santos, P.M.e.P.G.: A trusted infrastructure for symbolic analysis of event-driven web applications. European Conference on Object-Oriented Programming (ECOOP 2020) **166** (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.28
25. Gardner, P., Maffeis, S., Smith, G.: Towards a program logic for javascript. vol. 47, pp. 31–44 (01 2012). https://doi.org/10.1145/2103621.2103663
26. Gardner, P., Smith, G., Watt, C., Wood, T.: A trusted mechanised specification of javascript: One year on. vol. 9206, pp. 3–10 (07 2015). https://doi.org/10.1007/978-3-319-21690-4_1
27. Gavin Bierman, Martím Abadi, M.T.: Understanding typescript (2014). https://doi.org/10.1007/978-3-662-44202-9$_1$1

28. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javascript. pp. 126–150 (06 2010). https://doi.org/10.1007/978-3-642-14107-2_7
29. Hongki Lee, Sooncheol Won, J.J.J.C.S.R.: Safe: Formal specification and implementation of a scalable analysis framework for ecmascript (2012)
30. José Fragoso Santos, Petar Maksimović, G.S.P.G.: Javert 2.0: compositional symbolic execution for javascript. Proceedings of the ACM on Programming Languages **3**, 1–31 (01 2019). https://doi.org/10.1145/3290379
31. Maffeis, S., Mitchell, J., Taly, A.: An operational semantics for javascript. pp. 307–325 (12 2008). https://doi.org/10.1007/978-3-540-89330-1_22
32. Manuel Serrano, R.B.F.: Dynamic property caches: a step towards faster javascript proxy objects (2020). https://doi.org/10.1145/3377555.3377888
33. Park, D., Ştefănescu, A., Roşu, G.: Kjs: A complete formal semantics of javascript. ACM SIGPLAN Notices **50**, 346–356 (06 2015). https://doi.org/10.1145/2813885.2737991
34. Politz, J., Carroll, M., Lerner, B., Pombrio, J., Krishnamurthi, S.: A tested semantics for getters, setters, and eval in javascript. vol. 48, pp. 1–16 (10 2012). https://doi.org/10.1145/2384577.2384579
35. Ravi Chugh, David Herman, R.J.: Dependent types for javascript (2012). https://doi.org/10.1145/2398857.2384659
36. Roberto Amadini, Alexander Jordan, G.G.F.G.P.S.H.S.P.J.S.C.Z.: Combining string abstract domains for javascript analysis: an evaluation) (2017). https://doi.org/10.1007/978-3-662-54577-5$_3$
37. ŞtefănescuŞtefan CiobâcăRadu MereutaBrandon M. MooreTraian Florin ŞerbănuţăGrigore Roşu, A.: All-path reachability logic (2019). https://doi.org/10.23638/LMCS-15(2:5)2019
38. Satish Chandra, Colin S. Gordon, J.B.J.C.S.M.S.F.T.Y.C.: Type inference for static compilation of javascript (2016). https://doi.org/10.1145/2983990.2984017
39. Simon Holm Jensen, Anders Møller, P.T.: Type analysis for javascript (2009). https://doi.org/10.1007/978-3-642-03237-0$_1$7
40. Thiemann, P.: Towards a type system for analyzing javascript programs. vol. 3444 (04 2005). https://doi.org/10.1007/978-3-540-31987-0_28
41. Vineeth Kashyap, Kyle Dewey, E.A.K.J.W.K.G.J.S.B.W.B.H.: Jsai: a static analysis platform for javascript (2014). https://doi.org/10.1145/2635868.2635904