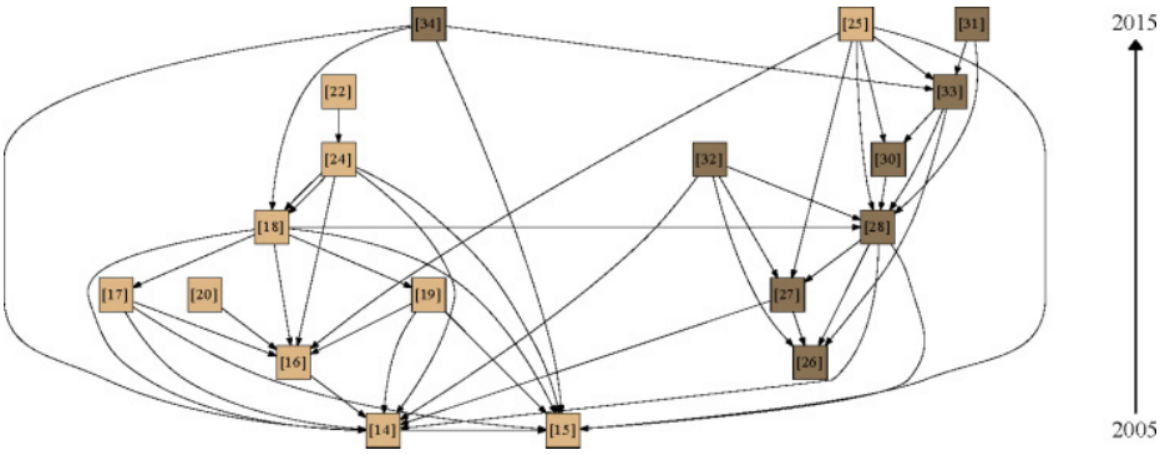
# Analiza Programelor JavaScript: Provocări și Trenduri de Cercetare

Recenzia lucrarii A59

## Introducere

Acest referat reprezinta recenzia lucrarii științifice “Analysis of JavaScript Programs: Challenges” [1]. Lucrarea are la bază 160 de alte lucrări științifice despre ecosistemul format în jurul limbajului JavaScript și provocările specifice. Aceste lucrări au fost grupate, prioritizate și analizate în baza unui graf de citatie. Astfel, cercetătorii au reușit sa grupeze lucrările similare și pe cele fundamentale din domeniu (cele care sunt citate cel mai des). Un exemplu de astfel de graf poate fi regasit în imaginea de mai jos pentru capitolul *Type Safety și Optimizarea JIT*.




Graf de citatii pentru Type Safety si Optimizarea JIT

Nevoia acestor cercetări este data atat de importanța și amploarea limbajului JavaScript, cât și de problemele des întâlnite în dezvoltarea programelor JavaScript. Chiar daca initial, JS a fost gandit doar ca un limbaj de scripting pentru web (1995), aplicabilitatea acestuia în zilele noastre este vast extinsă cuprinzand programe server-side, aplicatii web extrem de complexe, motoare de jocuri video, compilatoare și altele

In plus, programatorii prefera tot mai mult limbajul JS, fapt prezentat și de indexul TIOBE în care JS este constant în top 10 limbaje de programare. Parte din aceasta popularitate este data atat de expresivitatea limbajului JS, cat și de portabilitatea acestuia - practic poate fi rulat pe orice dispozitiv care poate rula un browser si/sau node.js

Aceasta expresivitate și flexibilitate este data în primul rând de faptul că JS este un limbaj dinamic. Obiectele și structurile de date pot fi modificate la runtime in orice mod, iar variabilele își pot schimba tipul de date in orice fel.

Pe de alta parte, aceasta flexibilitate poate duce foarte ușor la erori si comportament neasteptat. De asemenea, datorită naturii dinamice a limbajului, acesta este foarte greu de analizat. Si nu in ultimul rand, dat fiind faptul ca majoritatea programelor JS au la baza librării și cod terte, este foarte ușor sa introducem vulnerabilități de securitate.


# Caracteristicile Ecosistemului JavaScript

Limbajul JS a fost publicat în anul 1995 ca un limbaj de scripting simplu. Chiar dacă numele sugerează apropierea de limbajul Java, în realitate singurul motiv pentru care limbajul se numește așa este popularitatea limbajului Java din acea perioada. Astfel, creatorii au încercat sa atraga programatorii folosind aceasta sugestie și reușind sa il faca mai ușor de găsit.

În anul 1997 a fost publicată specificatia pe care se bazează limbajul JS, ECMAScript (ES). În următorii doi ani, alte două versiuni ale specificatiei au fost publicate. ES3 a pus bazele limbajului pe care il stim astazi.

Nevoia unei specificatii, în special pentru limbajul JavaScript, este dată de modul de implementare și rulare a limbajului și a programelor scrise cu acesta. Fiecare dezvoltator de browser poate propune propriul motor de rulare a programelor JS, atata timp cat respecta specificatia ECMAScript.

Următoarea versiune a ECMAScript, versiunea 5, a fost publicata in anul 2009. Aceasta a introdus, pe langa altele, module strict (*strict mode*) care dorea prevenirea folosirii constructelor predispuse la erori oferite de limbaj.

Mai apoi, in 2015, a fost publicata a sasea versiune, ES6 sau ECMAScript2015 - aceasta este nomenclatura corecta. ES2015 a adus o serie de imbunatatirii si caracteristici precum const/let si modules pe care le vom discuta in continuare.

# Caracteristicile Limbajului JavaScript

Dinamicitatea limbajului este data atat de sistemul dinamic de tipuri astfel incat orice variabila isi poate schimba tipul in cadrul executiei (aceasta duce deseori la comportament neasteptat datorat conversiilor precum conversia de la un string gol la `0` sau `false`), cat si de posibilitatea de a genera si rula cod in timpul executiei programul folosind `eval`.

```
get_cookie = function (name) {
  var ca = document.cookie.split(';');
  for (var i = 0, l = ca.length; i < l; i++) {
    if (eval("ca[i].match(/\\b" + name + "=/)"))
      return decodeURIComponent(ca[i].split('=')[1]);
  }
  return '';
}
```

Exemplu de folosire a functiei eval de pe fiverr.com

De asemenea, chiar daca limbajul JS poate fi considerat **object-oriented**, modul de definire ale obiectelor se bazeaza pe **prototype** si nu pe clase. Bineinteles, prototipul unui obiect este si el dinamic si poate fi modificat la runtime. Aceasta problema a fost vast diminuata prin introducerea **claselor** in ES2015 (la baza sunt doar syntatic sugar pentru prototype-based inheritance).

In realitate, JS este un limbaj multi-paradigm. Putem alege modul in care scriem cod sau putem combina mai multe moduri dintre procedural, orientat obiect si functional. Cel din urma este preferat in multe situatii, dar pana la introducerea modulelor in ES2015 modul de lucru era bazat pe **callbacks**. Acestea duc la o complexitate si mai mare de analiza a programului, deseori fiind mult mai greu de analizat decat programele scrise in C sau Java.

```
// Callback Hell


a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});

```

“Callback Hell”

## Caracteristicile Mediului Web

JavaScript este rulat de cele mai multe ori pe masina clientului in cadrul browser-ului si este gandit si interactioneze cu HTML-ul care confera structura paginilor web. Aceasta interactiune se realizeaza prin **Document Object Model** (DOM), o reprezentare arborescenta a documentelor HTML. Acest concept a fost extins de diverse framework-uri precum React care a introdus conceptul de **virtual DOM** mai apoi popularizat si de Angular 2+, Vue si Ember.
Pe langa uneltele pentru manipularea DOM-ului, fiind un limbaj gandit in primul rand pentru a gestiona interactiunea utilizatorului cu paginile web, exista si conceptul de **events**. Acesta se implementeaza prin functia `addEventListener` pentru anumite evenimente precum `onload` sau `onclick`. In plus, aceste evenimente sunt propagate elementelor ascendente (**bubble up**), iar de multe ori programatorii se folosesc de aceasta impreuna cu **event delegation** pentru a simplfica codul. De exemplu, daca avem un formular compus din mai multe campuri, putem adauga un event listener pe fiecare camp sau putem adauga un singur listener pe intregul formular care va fi invocat pentru evenimentele din toate campurile. Aceasta creste si mai mult complexitatea de analiza a codului.

Acest articol se concentreaza pe JS in mediul web, neincluzand partea server-side (Node.js).

## Erori și Vulnerabilități Specifice

Cele mai des intalnite erori in JS pot fi clasificate ca fiind the doua tipuri:

- `TypeError` - de exemplu, invocarea unei entitati care nu este functie duce la aceasta eroare
- `ReferenceError` - de exemplu, citirea unei variable care nu a fost declarata in prealabil.

Mai apoi, vulnerabilitatile de securitate pot fi catalogate astfel:

- **Cross-site scripting (XSS)** - rularea unui cod malitios care nu face parte din programul initial. Aici avem bine-cunoscutul exemplu de pe MySpace cu Samy is my hero [2]
- **Open redirect** - deseori bazat pe XSS si consta in navigarea nedorita pe un alt site, deseori malitios
- **Information leakage** - data fiind capacitatea JS de a trimite cereri catre un server, ne putem imagina situatii in care date sensibile sunt trimise nedorit la alte terte.
- **Code injection** - se bazeaza pe **eval**.

# Analiza Statică

O parte consederabila din lucrarile analizate dezbat analiza statica si scot in evidenta dificultatea acestea pentru JS date fiind cele discutate mai sus precum tipul dinamic, codul dinamic si obiectele dinamice.

Analiza statica porneste cu un subset de functionalitati ale limbajului, in timp ajungand sa acopere tot mai mult sau sa includa tool-uri care pot reduce din scop. Exemple pentru astfel de tool-uri includ *Unevalizer* si *Evalorizer*. Acestea functioneaza impreuna pentru a analiza

cazurile de utilizare ale functiei *eval* in cadrului unui program si a propune inlocuirea acesteia cu alternative mai bine definite bazate pe *string constants* si *JSON*.

Un alt subiect important este analiza librariilor. De exemplu pentru jQuery, poate cea mai folosita librarie de JS (chiar daca necesitatea ei a scazut dramatic in ultimii ani dupa introducerea ES2015 si framework-urilor consacrate de astazi), Schäfer propune in lucrarea sa intitulata “Dynamic Determinacy Analysis”[3] un mod de a analiza variabilele si functiile apelate cel mai frecvent si care isi pastreaza sau rezulta in aceeasi valoare. De fapt, acesta tehnica este folosita initial pentru analiza dinamic a programelor JS, iar mai apoi rezultatele obtinute sunt folosite pentru a scrie programe specializate pe cazurile de utilizare analizate. Mai apoi, se poate realiza analiza statica pe aceste programe.

O alta provocare pentru analiza codului JS este reprezentat de *loop*-uri. Pe langa varietatea acestora (*for-in, for-of*) avem si imbricarea acesta (nested loops) care creste exponential complexitatea analizei. O tehnice des intalnita este **loop unrolling** care practic presupune inlocuirea lor cu codul procedural si secvential aferent pentru un numar dat de iteratii. Aceasta tehnica este folosita si in cazuri extrem de sensibile cu privire la performanta.

Un compromis important cu privire la analiza statica este dat de echilibrul intre corectitudine (soundness) si scalabilitate. De exemplu, in cadrul unui IDE, scalabilitatea poate fi mai importanta deoarece programatorul trebuie sa primeasca rezultatul cat mai rapid si isi pot permite sa omita anumite sugestii.

Utilizarea rezultatelor produse de analiza statica sunt folosite pentru:

- **Detectarea erorilor de tip** - cu tool-uri precum TSCheck [4]
- **Detectarea vulnerabilitatolor de securitate** - cu tool-uri precum Gatekeeper care permitea implementarea de politici de securitate cu privile la ce este permis sa ruleze in cadrul programelor
- **Intelegerea programelor** - des folosit pentru refactorizare cu exemple precum JSRefactor [5]
- **Memory leaks** - un exemplu ar fi JSWhiz care a descoperit diferite scurgeri de memorie inclusiv in multiple produse Google

In practica astfel de unelte si cercetari de analiza static au dus la tool-urile si librariile pe care le folosim astazi precum **ESLint** si **Prettier**.

# Analiza Dinamică

In comparatie cu analiza statica car aproximeaza toate valorile posibile pentru un program, analiza dinamica foloseste tehnici precum **crawling** sau **dynamic symbolic execution**. Acestea au ca scot extinderea executiilor posibile ale programelor deoarece analiza dinamica clasica are la baza doar executia curenta.

Tehnicile de crawling colecteaza diferite stari ale documentului (ale DOM-ului mai exact) prin producerea de evenimente pana cand nu mai rezulta nicio stare noua.

Analiza dinamica a permis depistarea de **race conditions** precum si analiza de performanta si securitate ale programelor JS.

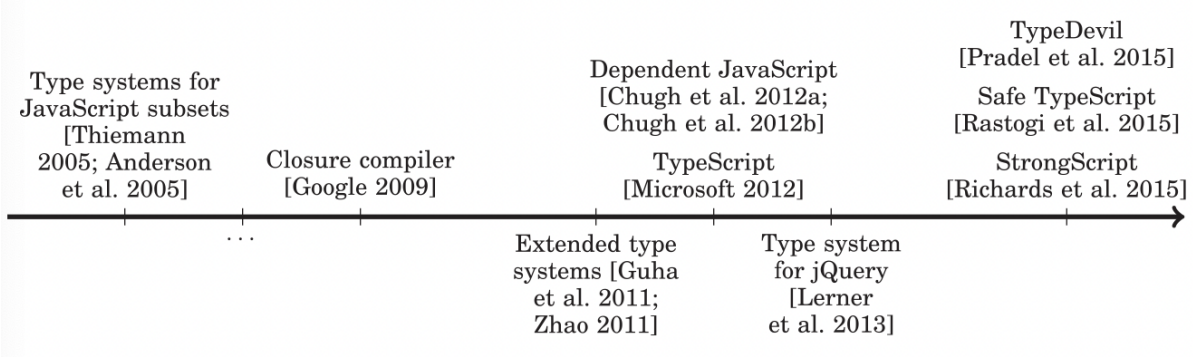
# Trenduri de Cercetare

## Formularizare si Rationament

Marea provocare aici rezulta chiar din specificatia limbajului. Din moment ce ECMAScript este in format text/proza, acesta poate fi interpretat in diverse moduri si poate omite cazuri exceptionale.

Cele mai notabile proiecte recente pentru formularizarea limbajului JS includ JSCert [6] si KJS [7]. Cel din urma a descoperit bug-uri inclusiv in V8 - motorul de JS din spatele browser-ului Chrome.

## Type Safety si Optimizarea JIT




Istoric al cercetarii pentru Type Safety

In 2005, Thiemann si Anderson au propus systeme de tipuri pentru un subset al functionalitatilor JS. Acestea se concetrau pe greseli frecvente si usor de inteles precum incercarea de apelare ale obiectelor non-functii si efectuarea de operatii matematice pe obiecte care nu sunt numere.

Mai apoi, Guha impreuna cu ceilalti autori au propus un sistem bazat pe fluxul de executie. Spre deosebire the primele sisteme, acesta putea detecta si obiectele nedefinite.

Poate cele mai marcante unelte publicate au fost **Closure**, un compilator creat de Google pentru analiza codului, minificarea si compilarea acestuia intr-o forma cat mai optimizata a limbajului bazata pe adnotari de tip precum si **TypeScript**, un superset al JavaScript care are o forma mult mai apropiata de limbajele statically typed precum Java sau C#.

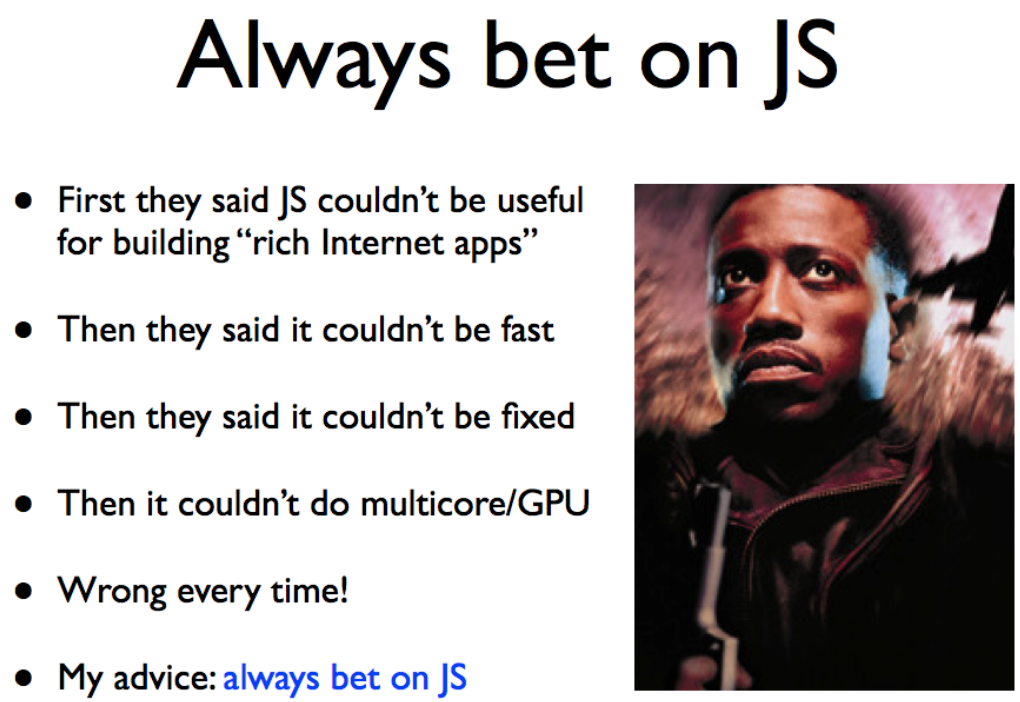
O provocare pentru dezvoltatorii TypeScript este reprezentata de librariile consacrate de JS care nu urmeaza sa fie rescrise in TypeScript in viitorul apropiat. Astfel, au aparut librarii aditionale cunoscute sub numele de DefinitelyTyped [9].

In legatura cu optimizarea JIT (just-in-time), cercetatorii au intampinat si aici dificultatii tot din cauza naturii dinamice ale limbajului, in special din cauza modificarii tipurilor variabilelor la runtime. JIT consta in generarea de cod optimizat pentru anumite parti ale programului (cele mai des executate). Problema este ca nu putem fi siguri ca o variabila va avea mereu acelasi tip. De exemplu, putem avea un array cu toate elementele avand acelasi tip mai

putin ultimul. Cand executia va ajunge la ultimul element, compilatorul va trebui sa renunte la codul generat, acest proces fiind unul foarte costisitor.

## Concluzii

In pofida complexitatii date de natura dinamica a limbajului si functionalitatilor necesare pentru functionarea in cadrului browser-elor (events, DOM etc.) si erorilor frecventa intalnite in productie, comunitatea JavaScript a continuat sa creasca si sa rezolva majoritatea problemelor. Gratie suportului oferit de marile companii precum Google, Facebook si Mozilla, ecosistemul a continuat sa creasca si experienta de dezvoltare sa fie tot mai buna.




# Referinte

[1] Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript Programs: Challenges and Research Trends. ACM Comput. Surv. 50, 4, Article 59 (July 2018), 34 pages. DOI:https://doi.org/10.1145/3106741
[2] https://samy.pl/myspace/tech.html
[3] Max Schäfer, Manu Sridharan, Julian Dolby, Frank Tip, *Dynamic Determinacy Analysis*, 2013, https://manu.sridharan.net/files/PLDI13Determinacy.pdf
[4] Feldthaus and Møller, *Checking Correctness of TypeScript Interfaces for JavaScript Libraries*, https://cs.au.dk/~amoeller/papers/tscheck/paper.pdf
[5] https://www.brics.dk/jsrefactor/
[6] https://jscert.org/

[7] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: a complete formal semantics of JavaScript. SIGPLAN Not. 50, 6 (June 2015), 346–356. DOI:https://doi.org/10.1145/2813885.2737991
[8] https://github.com/google/closure-compiler
[9] https://github.com/DefinitelyTyped/DefinitelyTyped