Operating Systems mutations, how and why integrate the user in the digital era ?

Etienne Brodu

May 9, 2015

Abstract

Abstract

Contents

Chapter 1

Introduction

Chapter 2

Context and Objectives

2.1 Javascript

2.1.1 Explosion of Javascript popularity

In the beginning

Javascript was created by Brendan Eich in 10 days in May 1995, while he was working at Netscape. The inital name of the project was Mocha, and then switched to LiveScript when released to the public in September 1995. The name Javascript was later adopter to leverage the trend around Java. Indeed, Java was considered the new hot web programming language at this time.

Microsoft released a concurrent implementation of Javascript in June 1996 in their browser Internet Explorer 3. They changed the name to JScript, to avoid trademark conflict with Oracle Corporation, who owns the name Javascript. But the differences between the two implementations made difficult for a script to be compatible for both. Netscape submitted Javascript to Ecma International for standardization in November 1996. In June 1997, Ecma International released ECMA-262, the first specification of EcmaScript, the standard for Javascript.

It was designed as a simple language to attract unexperienced developers. By opposition to Java or C++, which target professional developers.

Rising of the unpopular language

Javascript started as a programming language to animate web pages. It was used as a script language to implement short interactions. The main usage was for validate form on the client, avoiding unnecessary calls to the server.

In 2005 James Jesse Garrett released Ajax: A New Approach to Web Applications, a white paper coining the term Ajax. [**Garrett2005**] Ajax, consists in using Javascript to dynamically request and refresh the content of a web page. This paper point the trend in using this technique, and explain the consequences on user experience. Indeed, an application able to react to the user without completely refreshing the page, is bringing the web closer to a native application. At the time, some important web application started using this technique. The most important being Gmail, the google mail client.

At roughly the same time famous Javascript libraries were released with the goal to fill the differences between browsers implementations of Javascript.

Current situation : complete world domination

All modern web browsers now include a Javascript interpreter, making Javascript the most ubiquitous runtime in history. [**Flanagan2006**]

But the fact that Javascript is the language of the web, also made it famous in the open source communities. Javascript is the language counting the most repository on github, and is the most tagged language in StackOverflow.

Languages like Java and C/C++ are in active use in the software industry. Javascript on the other hand, is rising from the open source community and is slowly taking over the software industry as well.

With the release of HTML5, the importance of Javascript was completely acknowledged. It marked the consideration of the web as viable alternative for desktop clients. And with the mobile trends, most products are now released as a mobile apps, and a web application.

The web is now the norm. It is the application platform. And because Javascript is programming language of the web, it is *de facto* the language of choice to develop technologies.[1]

Some might consider HTML5 is not yet ready to build complete application on mobile, where condition in terms of performance, and accessibility are

[1] `http://blog.codinghorror.com/javascript-the-lingua-franca-of-the-web/`

not as good as on the desktop. But even in these cases, Javascript seems to remain the language of choice, as proven with React Native[2], from Facebook, who prone the "learn once, write anywhere", in opposition to the "develop once deploy everywhere of the web".

Isomorphic Javascript https://www.meteor.com/
https://facebook.github.io/flux/

Reactive http://facebook.github.io/react/
Code reuse. Why it never worked ?
https://www.destroyallsoftware.com/talks/the-birth-and-death-of-javascript
The Atom editor is written in Javascript node.js.
Now, major PaaS (which one) support node.js by default.
Heroku support Python, Java, Ruby, Node.js, PHP, Clojure and Scala
Amazon Lambda Web service support node.js in priority.
»> News :
npm raises 8m. http://techcrunch.com/2015/04/14/popular-javascript-package-manager-npm-raises-8m-launches-private-modules/

2.1.2 Overview of the language

Javascript was released in a hurry, without a strong and directive philosophy. During its evolution, it snowballed with different features to accommodate the community, and the usage it was made on the web. As a result Javascript contains various, and sometimes conflicting, programming paradigms. It borrow its syntax from a procedural language, like C, and the object notation from an object-oriented language, like Java, but it provides a different inheritance mechanism, based on prototypes. Most of the implementation adopt an event-based paradigm, like the DOM[3] and node.js[4]. And finally, event though it is not purely functional like Haskel, Javascript borrows some concepts from functional programming.

In this section, we focus on the last two programming paradigm, functional programming and event-based programming. Javascript exposes two

[2] `https://facebook.github.io/react-native/`
[3] `http://www.w3.org/DOM/`
[4] `https://nodejs.org/`

features from functional programming. Namely, it treats functions as first-class citizen, and allows them to close on their defining context, to become closures. We will explain these two features in details, and see how they are highly attractive to program in an event-based paradigm.

Functions as First-Class citizens

"All problems in computer science can be solved by another level of indirection"

—Butler Lampson

Javascript treats function as first-class citizens. One can manipulate functions like any other type (number, string ...). She can store functions in variables or object properties, pass functions as arguments to other functions, and write functions that return functions.

The most common usage examples of these features, are the methods `Map`, `Reduce` and `filter`. In the example below, the method `map` expect a function to apply on all the element of an array to modify its content, and output a modified array. A function expecting a function as a parameter is considered to be a higher-order function. `Map`, `Reduce` and `Filter` are higher-order functions.

```
1  [4, 8, 15, 16, 23, 42].map(function firstClassFunction(element) {
2    return element + 1;
3  });
4  // -> [5, 9, 16, 16, 24, 43]
```

Higher-order functions provide a new level of indirection, allowing abstractions over functions. To understand this new level of abstraction, let's briefly summarize the different abstractions on the execution flow offered by programming paradigms. In imperative programming, the control structures allow to modify the control flow. That is, for example, to execute different instructions depending on the state of the program. Procedural programming introduces procedures, or functions. That is the possibility to group instructions together to form functions. They can be applied in different contexts, thus allowing a new abstraction over the execution flow.

So, higher-order functions add another level of abstraction. It allows to dynamically modify the control of the execution flow. The ability to manipulate functions like any other value allows to abstract over functions, and behavior.

Higher-order functions replace the needs for some Object oriented programming design patterns.[5] Though object oriented programming doesn't exclude higher-order functions.

They are particularly interesting when the behavior of the program implies to react to inputs provided during the runtime, as we will see later. Web servers, or graphical user interfaces, for examples, interact with external events of various types.

Closures

"An object is data with functions. A closure is a function with data."

—John D. Cook

Closures are indissociable from the concept of lexical environment. To understand the former, it is important to understand the latter first.

Lexical environment A variable is the very first level of indirection provided by programming languages and mathematics. It is is a binding between a name and a value. Mutable like in imperative programming to represent the reality of memory cells, or immutable like in mathematics and functional programming. These bindings are created and modified during the execution. They form a context in which the execution takes place. To compartmentalize the execution, a context is also compartmentalized. A certain context can be accessed only by a precise portion of code. Most languages defines the scope of this context using code blocks as boundaries. That is known as lexical scoping, or static scoping. The variables declared inside a block represent the lexical environment of this block. These lexical environments are organized following the textual hierarchy of code blocks. The context available from a certain block of code, that is set of accessible variable, is formed as a cascade of the current lexical environment and all the parent lexical environment, up to the global lexical environment.

Javascript lexical environment [6]

Javascript implement lexical scoping with function definitions as boundaries, instead of code blocks. The code below show a simple example of

[5] `http://stackoverflow.com/a/5797892/933670`

[6] `http://www.ecma-international.org/ecma-262/5.1/#sec-10.2`

lexical scoping in Javascript.

```
var a = 4;
var c = 6;
function f() {
  var b = 5;
  var c = 0;
  // a and b are accessible here.
  return a + b + c;
}

f(); // -> 9

// b is not accessible here :
a + b + c; // -> ReferenceError: b is not defined
```

Javascript is a dynamic language. So the lexical scope provided by Javascript is dynamically modifiable by two statements : `with` and `eval`.

A closure is the association of a first-class function with its context. When a function is passed as an argument to an higher-order function, she closes over its context to become a closure. When a closure is called, it still has access to the context in which it was defined. The code below show a simple example of a closure in Javascript. The function `g` is defined inside the scope of `f`, so it has access to the variable `b`. When `f` return `g` to be assigned in `h`, it becomes a closure. The variable `h` holds a closure referencing the function `g`, as well as its context, containing the variable `b`. The closure `h` has access to the variable `b` even outside the scope of the function `f`.

```
function f() {
  var b = 4;
  return function g(a) {
    return a + b;
  }
}

var h = f();
// b is not accessible here :
b; // -> ReferenceError: b is not defined

// h is the function g with a closure over b :
h(5) // -> 9
```

2.1.3 Turn-based programming

Javascript, like most programming languages, is synchronous and non-concurrent. The specification doesn't provide mechanism to write concurrent execution of Javascript. There is no reference to `setTimeout` nor `setInterval`. These two well-known instructions to asynchronously post-pone execution are provided

by the DOM. Indeed, like for many languages, concurrency is supported and provided by the execution engine. For example the JVM in the case of Java, or the operating system in the case of C/C++. These last two languages were mostly used to design CPU intensive applications. The concurrency model for such application is driven by the need for computing power. The best concurrency model is the threading model. It allows to run multiple executions simultaneously to leverage the potential of parallel architectures, and execute faster. Execution engine provide thread libraries to allow concurrency, like pthreads[7] for POSIX operating systems, and the `Thread`[8] class for Java. But, as we will see in a next chapter, threads are known to be very difficult to manipulate. It is lucky that Javascript was not seen early as a language to build CPU intensive applications, so it can adopt a different concurrency model.

Indeed, Javascript was used from the beginning to build graphical user interfaces. Since user interfaces evolved from a simple and sequential user prompt to a full interactive graphical space, the programs behind such interface need to display and process concurrent events.

As the graphical space to render an application is inehrently concurrent, GUIs need to react to multiple events occurring concurrently. The concurrency need is completely different than for CPU intensive applications, and so is the concurrency model. Graphical user interfaces often use an event-loop.

As said above, most implementations of Javascript feature an event-based programming paradigm. The DOM[9] and node.js[10] are the most famous examples.

HOF allows Continuation Passing Style, which is particularly useful in a turn-based programming language such as node.js javascript.

the Javascript event-loop

Web pages are graphical environment offering multiple area of interaction for the user. Because of this multiplicity, the traditional linear programing model doesn't hold anymore. Graphical systems switched from this linear programming model to a different programming model focused on events.

[7] `https://computing.llnl.gov/tutorials/pthreads/`
[8] `https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html`
[9] `http://www.w3.org/DOM/`
[10] `https://nodejs.org/`

Javascript uses higher-order functions. It is the ability for a language to manipulate functions like any other value. This ability is used to register a function to trigger after an event occurred. An event might be the click on an element of the page, for example.

Such a function is named a callback, a handler, a listener ... And it shift the programming paradigm from synchronous to asynchronous, which is a big deal.

In synchronous programming, the computation step are executed sequentially, one after the other. The program execution follows perfectly the program layout written in a linear textual file.

On the other hand, asynchronous programming allows a step back from this linearity.

A multi-threaded system allows the developer to explicitly express the parallelism in the application. A GOTO statement allows the developer to explicitly express the control flow in the application.

Asynchronous programming allows the program to manage the concurrency of the execution. Unlike a linear layout of an imperative program, it allows to express more finely the dependencies between instructions.

Promises

2.2 Scalability

We define a web service as a computer program whose main interface is based on web protocols, such as HTTP. Such a service uses resources allocated on a network of computers. Scalability defines the ability of the service to use a certain quantity of resource to meet a desired performance. We call system the association of the computer program and the available resources. The performance of this system is measured by its latency and throughput.

2.2.1 Latency and throughput

Latency is the time elapsed between the reception of a request, and the sent of the reply. It includes the time waiting for resources to be free to process the request, and the time to process the request.

Throughput is the number of requests processed by the system by unit of time.

Latency and throughput are linked in a certain way. If a modification of the web service reduces its mean latency to a half, then the throughput doubles immediately. It takes half the time to process a request, therefore, the service can process more requests in the same time. However, if throughput augment, the latency doesn't necessarily decrease.

2.2.2 Scalability granularity

We define a computer program as a set of operations. In the case of a web service, these operations can be directly requested by the user through the interface. An operation can cause any other operation to execute.

Because both the resources used and the operations executed are discrete : not infinitely divisible, scalability is inherently discrete.

Scalability granularity is the increment of resources. How the input data can be split up ? How the program can be deployed on many machines ?

We call system the association of the computer program with the resources

2.2.3 Horizontal and vertical scaling

There is two ways to augment the resources of the system. Enhance the nodes in the computer network - vertical scaling. Or add more nodes to the computer network - horizontal scaling.

There are three theories, from the most restrictive, to the most general.

Scalability is the property of a computer program to occupy available resources to meet a needed performance. EIther in Latency, or in throughput.

2.2.4 Linear scalability

Clements et. al. [**Clements2013a**] prove that a computer program scale linearly if all its operations are commutative. Two operations are said to be commutative if they can be executed in any orders, and the same initial state will result in the same final state. Commutativity implies the two operations to be memory-conflict free, or independent, which is equivalent to say that they can be executed in parallel.

Therefore, to achieve linear scalability, a computer program must be composed of a set of operations that commutes. Thus, all the operations are

parallel, they can be executed simultaneously, on any number of machines as required.

The size of the operations sets the scalability granularity.

However, commutativity is not achievable in real applications. Even sv6, the operating system resulting from the work on commutative scalability only has 99% commutativity. For real application, in the best case, the granularity is coarse, in the worst case, there is no possible commutativity because of shared resources (like a product inventory, or a friend graph).

2.2.5 Limited scalability

Amdahl introduced in 1967 a law to predict the limitation of speedup a computer program can achieve if a fraction of its code is sequential. Amdahl worked at increasing the speed of computer clock, while the scientific community was working on improving parallelism of computing machines.

In a set of operations, even if one is non-commutative, it cannot be executed in parallel of any others, the scalability is limited by this operation.

There is a difference if the operation is non-commutative with itself, or only with others. In the first case, it impose a queuing, while in the second case, it only increase the granularity : you can regroup the non-commutative operation with its subsequents, and form a bigger commutative operation.

2.2.6 Negative scalability

Gunther generalized Amdahl's law into the Universal Scalability Law. It includes the parallelization of non-independent operations with the use of synchronization.

It models the negative return on scalability from sharing resources observed in many real world applications.

2.2.7 Eventual Consistency

To overpass the scalability limits set by the previous rules, it is possible to abandon consistency. It simply tolerate incoherences between multiple replicas. The output of an operation can be false while its state is synchronized with the other replicas.

2.3 Concurrency

2.3.1 About concurrent systems

This demonstration focus on application depending on long waiting operations like I/O operations. Particularly, this demonstration focus on real-time web services. It is irrelevant for application heavily relying on CPU operations, like scientific applications.

Threads-based system and event-based system evolved significantly over the last half century. These evolutions were fueled by the long-running debate about which design is better. We try to succinctly and roughly retrace theses evolutions to understand the positions of each community. This demonstration show that thread and events are two faces of the same reality.

Lauer and Needham [**Lauer1979**] presented an equivalence between Procedure-oriented Systems and a Message-oriented Systems.

Adya *et. al.* analyzed this debate and presented fives categories through which to present the problem [**Adya2002**]. These two categories were often associated with thread-based systems and event-based systems. Their advantages and drawbacks were mistaken with those of thread and events. Adya *et. al.* explain in details two of these categories that are most representative, Task management and Stack management. We paraphrase these explanations.

Task management

Consider a task as an encapsulation of part of the logic of a complete application. All the task access the same shared state. The Task management is the strategy chosen to arrange the task executions in available space and time.

Preemptive task management executes each task concurrently. Their executions interleave on a single core, or overlap on multiple cores. It allows to leverage the parallelism of modern architectures. This parallelism has a cost however, developers are responsible for the synchronization of the shared memory. While accessing a memory cell, it must be locked so that no other task can modify it. Synchronization mechanism impose the developer to be especially aware of race condition, and deadlocks. These synchronization problems make concurrency hard to program with preemptive task management.

The opposite approach, Serial task management, executes each task to completion before starting the next. The exclusivity of execution assures an exclusive access on the memory. Therefore, it removes the need for synchronization mechanism. However, this approach is ill-fitted for modern applications, where concurrency is needed.

A compromise approach, Cooperative task management, allows tasks to yield voluntarily. A task may yield to avoid monopolizing the core for too long. Typically, it yields to avoid waiting on long I/O operations. It merges the concurrency of the preemptive task management, and the exclusive memory access. Thus, it relieves the developer from synchronization problems. But at the cost of dropping parallel execution.

Threads are associated with preemptive task management, and events with Cooperative task management. For this reason, it is commonly believed that synchronization mechanisms make threads hard to program [**Ousterhout1996**]. While it is really Preemptive task management that is responsible for these synchronization problems [**Adya2002**].

Stack management

Consider a task is composed of several subtasks interleaved with I/O operations. Each I/O operation signal its completion with an event. The task stops at each I/O operation, and must wait the event to continue the execution. The stack management is the strategy chosen to express the sequentiality of the subtasks.

The automatic stack management is what is mostly used in imperative programming. The execution seems to wait the end of the operations to continue with the next instruction. The call stack is kept intact. This is what is commonly called synchronous programming.

In the manual stack management, developers need to manually register the handlers to continue the execution after the operation. The execution immediately continues with the next instruction, without waiting the completion of the operation. It implies to rip the call stack in two functions; one to initiate the operation, and another to retrieve the result. This is what is commonly called asynchronously programming.

What we argue is that synchronous is good because it is linear, it avoids stack ripping. But asynchronous is good because it allows parallelism by default.

Threads are associated with the automatic stack management, and events

with manual stack management. For this reason, it is commonly believed that threads are easier to program. [**Thread systems allow programmers to express control flow**] [**Behren2003**]. However, the automatic stack management is not exclusive to threads. Fibers, presented by Adya *et. al.* is an example of cooperative task management with automatic stack management [**Adya2002**] . Fibers present the advantage of cooperative task management, without the disadvantage of stack ripping. That is the ease of programming because of the absence of synchronization, without the difficulty of stack ripping.

We argue that the advantages of manual stack management outweigh its drawbacks for web services. Because of the numerous I/O operations, parallelism is

But what is actually highlighted is the automatic state management provided by threads. And with lighter context change, threads are a good choice which provide parallelism.

Historically, events-based system are associated with manual state management, while threads-based systems are associated with automatic state management. Manual state management imposed stack ripping [**Adya2002**]. With closure, it is not the case anymore. Events now have automatic state management as well [**Krohn2007**].

Now, there is implementation of thread model with cooperative management, with context-switch overhead improved enough to fill the gap with events model. And there is implementation of event model with automatic state management filling the gap with thread model. In this condition, we ask, what really is the difference between thread and events. We argue there is none. Except the isolation, versus sharing of the memory, which, again is not significant of either. In the first case, the different execution threads exchange messages, while in the second, they use synchronization mechanism to assure invariants in their states

For a single thread of execution, both model could avoid synchronization through cooperative task management, which assure invariants. Or avoid procedure slicing (if any) using synchronization. These are the two ends of a design spectrum. One end (cooperative task management) fits better for small processing with heavy use of shared resources. While the other end (synchronization) fits better for long processing with small use of shared resources. When one end of the design spectrum is used while the other should be used, one might expect unresponsiveness because of too heavy events, or performance fall due to interlocking.

Scalability is achieved through parallelism, which is itself achieved in our

case (web servers) through cluster of commodity machines.

With distribution, this design spectrum gets a better contrast.

The synchronization of distributed, shared resources is limited through the CAP theorem [**Gilbert2002a**]. Partition tolerance is a requirement of a distributed system. One needs to choose good latency (availability) or consistency. The CAP theorem is generalized into a broader theroem about[**Gilbert2012**]

The isolation of resources implies to split the architecture in different stages, like Ninja [**Gribble2001**], SEDA [**Welsh2000**], or Flash [**Pai1999**]. This splitting is difficult for the developer. The splitting which is good for the machine, is not the same as the one good for the design in modules.

The two ends of this design spectrum presented map directly onto the two kinds of parallelism advocated for scalability. That is pipeline parallelism, and data parallelism.

Pipeline parallelism is good for data locality, and important throughput. But each stage adds an overhead in latency.

Data parallelism is good for latency, because one request is processed from beginning to the end without waiting in queues. But it implies that the different machines share a common database. Which is a shared resource, and is limited by the CAP theorem.

Both parallelism have advantages and drawbacks, and both could be combined, like in the SEDA architecture. Ultimately, it would be possible to design a design spectrum to choose which kind of parallelism for a set of requirements. But we leave this for future works.

Splitting an architecture in stages is a difficult process, which prevent future code refactoring, and module modifications. We argue that the design for the technical architecture, and the design for the human minds should not be the same. Threads belong to the mental model, the design granularity Events belong to the execution model, the architecture granularity It is a mistake to attempt high concurrency without help from the compiler [**Behren2003**]. Through compilation, we want to transform an event-loop based program (cooperative task management, no synchronization) into a pipeline parallelism distributed system. So, basically, we argue that it is possible to distribute one loop event onto multiple execution core.

/! WARNING The paper Why events are a bad idea states that : the control flow patterns used by these applications fell into three simple categories: call/return, parallel calls, and pipelines. Indeed, it is no coincidence that common event patterns map cleanly onto the call/return mechanism of

threads. Robust systems need acknowledgements for error handling, for storage deallocation, and for cleanup; thus, they need a "return" even in the event model. » Why is it completly false ? It is crucial to find an answer. Moreover, Ayda et. al. state that : For the classes of applications we reference here [file servers and web servers], processing is often partitioned into stages. Other system designers advocated non-threaded programming models because they observe that for a certain class of high-performance systems [...] substantial performance improvements can be optained by reducing context switching and carefully implementing application-specific cache-conscious task scheduling.

The paper Why events are a bad idea states that : One could argue that instead of switching to thread systems, we should build tools or languages that address the problems with event systems (i.e., reply matching, live state management, and shared state management). However, such tools would effectively duplicate the syntax and run-time behavior of threads. » Well, yes ... With the exception of the stack junction. The paper on Duality had it right, their graph is correct, but for threads, it cannot be distributed because of stacks, while for events, it can.

Software evolution substantially magnifies the problem of function ripping: when a function evolves from being compute-only to potentially yielding, all functions, along every path from the function whose concurrency semantics have changed to the root of the call graph may potentially have to be ripped in two. (More precisely, all functions up a branch of the call graph will have to be ripped until a function is encountered that already makes its call in continuation-passing form.) We call this phenomenon "stack ripping" and see it as the primary drawback to manual stack management. Note that, as with all global evolutions, functions on the call graph may be maintained by different parties, making the change difficult. » Stack ripping is what I am talking about. While the stack are joined, it is not possible to distribute. If they say that stack ripping is necessary, that means it is not possible to encapsulate asynchronous function into synchronous function.

2.4 Objectives

2.4.1 LiquidIT

Development Scalability

Performance Scalability

2.4.2 Proposal and Hypothesis

Chapter 3

State of the Art

3.1 Flow-programming

3.2 ... ?

Chapter 4

Pipeline parallelism for Javascript

From here, the reader should be confortable with the event-loop, and the analogy we drawn between the event-loop and a pipeline. The problematic is now clear : how to split the heap so that each asynchronous callback has its own exclusive heap ?

4.1 Callback isolation

We explain in this section the compilation process we developped to isolate the memory access for each callbacks. The result of this process should be two-fold. First each callback should have an exclusive access on a region of the memory. So that two different callback can be executed in parallel. And it should be clear for each callback, what are the variable needed from upstream callbacks, and what are the variable to send downstream.

4.1.1 Propagation of variables

Scope identification

In section **??**, we explained that Javascript is roughly lexically scoped. A consequence is that the declaration of contexts can be inferred statically. For example, in a lexically scoped, strongly typed, compiled language, the compiler know the content of each scope during compile time, and can prepare the memory stack to store the variables in each scope.

In most languages, the memory is in two parts : the stack, and the heap. The stack is statically scoped, and its layout is known at compile time. The

heap, on the other hand is dynamically allocated. Its layout is built at run time.

But Javascript is a dynamic language, perhaps the most dynamic of all languages. It doesn't have this distinction between stack and heap. Every variable is dynamically allocated on the heap. That induce two consequences. The first is that Javascript provides two statements to dynamically modify the lexial scope : `eval` and `with`. The second is that to know the layout of the heap, we need to use static analysis tools. In the next two sections, we adress these two consequences.

Break the lexical scope

Without these statements, `eval` and `with`, Javascript is lexically scoped. It is possible to infer the scope of each variable at compile time.

However, even if Javascript is lexically scoped, the memory is still dynamicall allocated and manipualeted, so that it is not possible to actually infer the memory layout at compiler time only with lexical scope analysis, and without deeper static analysis.

Scope Leaking

To infer the layout of the heap at compile time, static analysis tools are used, like the points-to analysis, developped by Andersen in its PhD thesis [**Andersen1994**]. For such analysis, the memory is splitted at the access scale. In low-level languages, like C/C++, the memory is mainly managed by the developer. Allowing access to the memory at a small grained scale : up to the address. It impose the analysis to split the memory to the adress scale in some cases. In higher-level languages, like Javascript, the developer cannot access the memory to the adress scale. The memory is accessed at a coarser scale : the property scale. (At the exception of some arrays and buffers, that mimic, and are mapped to actual memory adresses for performance reasons.)

Propagation of execution and variables

Chapter 5

Conclusion

.1 Language popularity

.1.1 PopularitY of Programming Languages (PYPL)

[1] The PYPL index uses Google trends[2] as a leading indicator of the popularity of a programming language. It search for the trend for each programming language by counting the number of searches of this language and the word "tutorial".

PYPL for May 2015

[1] http://pypl.github.io/PYPL.html
[2] https://www.google.com/trends/

Rank	Change	Language	Share	Trend
1		Java	24.1%	-0.9%
2		PHP	11.4%	-1.6%
3		Python	10.9%	+1.3%
4		C#	8.9%	-0.7%
5		C++	8.0%	-0.2%
6		C	7.6%	+0.2%
7		Javascript	7.1%	-0.6%
8		Objective-C	5.7%	-0.2%
9		Matlab	3.1%	+0.1%
10	2× ↑	R	2.8%	+0.7%
11	5× ↑	Swift	2.6%	+2.9%
12	1× ↓	Ruby	2.5%	+0.0%
13	3× ↓	Visual Basic	2.2%	-0.6%
14	1× ↓	VBA	1.5%	-0.1%
15	1× ↓	Perl	1.2%	-0.3%
16	1× ↓	lua	0.5%	-0.1%

.1.2 TIOBE

[3]

The TIOBE index uses many search engines as an indicator of the current popularity of programming languages. It count the number of pages each search engine find when queried with the language name and the word "programming". This indicator indicates the number of resources available, and the discussions about a given programming language.

Javascript was the most rising language of 2014 in the TIOBE index.

TIOBE for April 2015

[3] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Apr 2015	Apr 2014	Change	Programming Language	Ratings	Change
1	2	↑	Java	16.041%	-1.31%
2	1	↓	C	15.745%	-1.89%
3	4	↑	C++	6.962%	+0.83%
4	3	↓	Objective-C	5.890%	-6.99%
5	5		C#	4.947%	+0.13%
6	9	↑	JavaScript	3.297%	+1.55%
7	7		PHP	3.009%	+0.24%
8	8		Python	2.690%	+0.70%
9	-	2× ↑	Visual Basic	2.199%	+2.20%

.1.3 Programming Language Popularity Chart

[4]

The programming language popularity chart indicates the activity of a given language in the online communities. It uses two indicators to rank languages : the number of line changed in github of, and the number of questions tagged with a certain language.

Javascript is ranked number one in this index. The Javascript community is particularly active online, and in the open source.

indeed.com

[4] http://langpop.corger.nl