CEPUS

Zeste de savoir

Quelques rouages d'un moteur JavaScript

12 août 2019

Table des matières

Il y a fort à parier que bon nombre de lecteurs ici n'ont pas connu les joies du web au siècle dernier. Les pages perso sur Geocities et Lycos, pleines de DHTML. Je parle de DHTML parce que c'était la technologie des débutants, les vrais faisaient du Macromedia Flash ou des composants ActiveX. Ha ha ha, c'était le bon temps. Non. NON. C'était affreux.

Les connexions étaient lentes à charger des sites moches et lents qui s'exécutaient lentement sur des machines lentes. Aussi, personne ne prédisait un grand avenir à JavaScript. Difficile de donner tort à ses détracteurs d'alors, d'ailleurs. Un langage torché en 2 semaines, censé être le Lisp du web mais en fait devenu le euh, le JavaScript du web ?

Personne n'y accordait une grande importance, donc. Et comme je le disais, on avait Flash et c'était bien mieux. Puis un jour Google a été fondé, et Google s'investissant à fond dans le web, Google a misé entre autres sur le JavaScript.

Entre 2007 et 2008, Google a donc décidé de se doter d'un navigateur, engageant de gros moyens financiers afin de créer un moteur JavaScript de nouvelle génération[1] : V8. Je parle d'argent, mais la majeure partie du succès de ce projet est très probablement due à l'équipe d'ingénieurs ayant posé les bases et implémenté la chose.

Quand à la fin de l'été 2008 la première version de Chromium sortait, on était face au moteur JavaScript le plus puissant, et de très loin. Il y avait largement de quoi écraser la compétition. Mais Chromium était open-source (c'est chouette que certains chez Google savaient que la compétition et le partage c'est bien), et ça a lancé une bataille marrante, un échange d'idées et de techniques, si bien qu'en quelques années et bien que V8 continue d'évoluer la concurrence avait rattrapé V8 et qu'actuellement tout le monde est environ au coude à coude.

JavaScript est désormais un langage incontournable. JavaScript est désormais également très performant.

Je vais ici parler brièvement de comment **V8, le moteur JavaScript de Google** Chrome, traite le JavaScript. Je vais ensuite développer un concept en particulier, les classes cachées ou *Map*s. Il y aurait bien d'autres sujets à traiter quand on parle de compilation ou de JavaScript ou, pire, de compilation de JavaScript. Ne vous étonnez pas si certains concepts ne sont mentionnés qu'en passant. Les développer individuellement prendrait parfois un article de longueur équivalente.

Mise à part cette introduction (et quelques références stupides et jeux de mots non moins stupides), cet article est une traduction / compilation de deux articles publiés en langue anglaise sur mon blog : https ://draft.li/blog/ . Les références et sources ont été laissées telles quelles

parce que je ne connais pas de littérature francophone à ce sujet. Pour cette raison, les liens sur lesquels vous cliquerez dans cet article risquent fort d'être en anglais.

1. Ouvrons le capot, décrivons les trucs qui brillent

Comme nous nous intéressons à l'infrastructure de compilation JavaScript du moteur V8, commençons par une vue d'ensemble des principaux éléments, partie qui servira également à établir quelques définitions.

1.0.1. Chromium, Chrome, Node.js

Le *Chromium Project*[1] chapeaute le développement de Chromium. Chromium, sorti en 2008, est le navigateur *open-source* sur lequel Google Chrome est basé. Le moteur JavaScript de Chromium se nomme V8[2]. D'autres projets utilisent le moteur V8, par exemple le navigateur Opera et Node.js.

1.0.2. V8

V8 compile du JavaScript en langage machine et exécute ce code immédiatement. Puisque lorsque V8 arrive sur nos bureaux la majorité des autres moteurs JavaScript sont des interpréteurs , il y a là une différence notable. Pour tirer un parallèle, ce que fait V8 est plus proche de ce que fait GCC avec un programme C que ce que fait CPython .

1.0.3. Crankshaft et ses *bailouts*

En 2010, le *Chromium Project* a publié une nouvelle version de V8 utilisant une nouvelle infrastructure de compilation nommée *Crankshaft*.

Les trois composants principaux de Crankshaft sont :

1. le ***base compiler*** (”compilateur de base”), un JIT qui compile JavaScript vers du langage machine le plus vite possible sans même tenter d'optimiser le code lors de la compilation,
2. le ***runtime profiler*** (”profileur d'exécution”), qui enregistre — entre autres — combien de temps est passé dans quelles parties du code et identifie le *hot code* (”code chaud”), c'est à dire le code qu'il vaudrait la peine d'optimiser en priorité, et
3. l'***optimizing compiler***, un JIT qui tente d'optimiser le *hot code* précédemment identifié.

1. Bientôt 10 ans après, l'informatique évoluant rapidement, les plans de 2008 ne sont plus vraiment la ”nouvelle génération”.

Il y a donc deux compilateurs. Ce sont des JIT .

L'intérêt d'un JIT est notable dans le cas de langages dynamiquement typés. Connaitre les types des variables permet beaucoup d'optimisations lors de la compilation. Le langage étant dynamiquement typé, il n'est pas possible d'avoir des informations complètes sur les types avant d'exécuter le code. Une belle possibilité offerte par un JIT est de compiler le code, l'exécuter tout en observant les types lors de l'exécution, puis de recompiler le code — toujours à la volée — en l'optimisant grâce au feedback de types récolté lors des exécutions précédentes du code.

Une optimisation est un compromis. Nous voulons à la fois des pages qui s'affichent rapidement et du code qui s'exécute rapidement, donc **un temps de démarrage rapide et des performances élevées**. Le démarrage rapide est ici rendu possible grâce au *base compiler* : V8 compile et exécute le code sitôt qu'il l'a à disposition. Les performances élevées sont dues à l'*optimizing compiler* : Crankshaft optimise le code qui vaut la peine d'être optimisé.

Optimiser tout le JavaScript avant de l'exécuter une première fois est une fausse bonne idée, notamment pour les deux raisons suivantes.

— L'optimisation n'est pas "gratuite", elle augmenterait donc le temps qu'un visiteur doit attendre avant de pouvoir interagir avec une page web lors de son premier chargement.
— Il est nettement plus facile d'optimiser du code qu'on a déjà exécuté. C'est là que le travail du *runtime profiler* devient utile : il va être une précieuse source d'informations pour l'*optimizing compiler*, lui disant qu'est-ce qui est de quel type, quelle fonction prend beaucoup de temps, quelle fonction est appelée très souvent, etc.

Lorsque l'*optimizing compiler* se met au boulot, il fait des hypothèses sur comment le code sera exécuté. Ses hypothèses sont optimistes : il part du principe que presque tout est optimisable, et il se dit que l'histoire se répète, même s'il s'avère plus tard que ce n'est pas le cas. Pour illustrer cette dernière proposition, imaginez que `f(x)` a été appelée avec `x = 13` et `x = 27`. Une hypothèse d'optimisation optimiste sera d'optimiser `f(x)` comme si `x` était absolument toujours un nombre.

Dans certains cas, les données (par exemple les informations de types) récoltées lors de l'exécution - le résultat du travail du *runtime profiler*, si vous avez bien suivi - données mises à disposition de l'*optimizing compiler*, se révèlent insuffisantes. Pour revenir à notre exemple, *paf*, une condition rarement exécutée mène à l'appel `f('bim')`. L'hypothèse d'optimisation n'était pas bonne ! L'*optimizing compiler* doit se résoudre à une ***deopt*** (désoptimisation) et renvoyer l'exécution de `f` au *base compiler*, qui certes ne fera pas de miracles niveau performances mais offre au moins l'avantage de savoir exécuter `f` quoi qu'il arrive.

Plus tard, si `f` est à nouveau appelé, l'*optimizing compiler* s'y penchera à nouveau, fort de nouvelles informations, et tentera une fois de plus d'optimiser cette fonction. Ses hypothèses seront plus précises, et peut-être arrivera-t-il à correctement optimiser `f`. Un nouvel échec signifie recommencer comme précédemment. Après 10 tentatives infructueuses, l'*optimizing compiler* décidera d'un *bailout* ("désistement"), il se désistera de toute tâche d'optimisation de cette fonction, disant simplement : "Optimized too many times3" ("Optimisé trop de fois").

Ceci n'est qu'un exemple de désistement. Notre compilateur se désiste également lorsqu'une fonction contient des constructions ou fonctionnalités de JavaScript qu'il ne supporte pas, ou lorsque le code ne respecte pas certaines limites arbitraires5. Dans ces deux derniers cas, le compilateur ne tente même pas d'optimiser le code. Il voit immédiatement qu'il vaut mieux

qu'il ne tente pas et dit "non", se désistant aussitôt, renvoyant donc l'exécution de ces parties du code au *base compiler*.

Heureusement, chaque désistement est accompagné d'un message expliquant la raison pour laquelle le *bailout* s'est produit. Un certain nombre de ces raisons sont documentées ici : *V8 bailout reasons*[4], le but de ce projet étant de montrer comment reproduire ces désistements, d'expliquer pourquoi Crankshaft se désiste et comment éviter ces situations.

Un peu de lecture supplémentaire à ce sujet :

- Première présentation de Crankshaft par l'équipe de V8
- Un ticket consacré aux *bailouts* sur le bug tracker des devtools de Google Chrome
- Une page de wiki expliquant ce qui "tue" les performances de V8 et comment éviter ces problèmes

```
1

2

3

4

5
```

2. Pédagogie des Maps

Quand Lars Bak[3], ingénieur en chef de V8, décrit[8] les décisions de conception à la base de V8, la première chose qu'il mentionne sont les *hidden classes*, littéralement "classes cachées" ou "*Maps*". (J'utiliserai principalement les termes "*Map*" / "*Maps*" dans le reste de l'article.)

Pour bien comprendre les *Maps*, il faut d'abord comprendre le problème qu'elles résolvent. Cette partie est une introduction à ce concept et va, dans un premier temps, expliquer ce qu'on entend par *fast property access* ou "accès rapide aux propriétés", parce que c'est ce que les *Maps* rendent possible. Dans un deuxième temps, j'expliquerai ce que sont ces *Maps* et d'où elles viennent.

2.0.1. Propriétés dans un monde OO

On pourrait diviser les langages orientés-objet en deux catégories : ceux qui ont des classes (Java, C++, Ruby, … : *class-based*) et ceux qui n'en ont pas (JavaScript, … : *prototype-based*). Plus intéressant peut-être, on peut faire la distinction entre les langages dans lesquels les classes et l'héritage sont entièrement connus lors de la compilation (Java, C#, C++, …) et ceux dans lesquels les classes (éventuelles) et l'héritage ne sont connus qu'à l'exécution (Python, JavaScript, Smalltalk, …).

1. https ://en.wikipedia.org/wiki/Chromium_(web_browser)
2. https ://en.wikipedia.org/wiki/V8_(JavaScript_engine)
3. https ://github.com/vhf/v8-bailout-reasons#optimized-too-many-times
4. https ://github.com/vhf/v8-bailout-reasons
5. https ://draft.li/blog/2016/01/15/one-simple-trick-for-javascript-performance-optimization/

Une différence notable est que dans le premier cas, il n'est pas possible d'ajouter des méthodes ou des propriétés à un objet ou à une classe lors de l'exécution. Ça permet de représenter nos objets de sorte qu'accéder à leurs propriétés est particulièrement efficace et peu coûteux.

Notez que JavaScript s'est depuis peu mis à utiliser le mot-clé `class` comme sucre syntaxique pour déclarer des prototypes…

Figure : … mais c'est pas ça qu'on appelle la classe, mon p'tit José.[4] (*Illustration © Warner Bros.*)

Pour ce faire on peut imaginer une "*Map*" commune à toutes les instances d'une classe, indiquant les adresses mémoire des données d'un objet. Accéder à une propriété d'un objet peut donc être fait en une seule instruction basique : "charge le truc situé à cette adresse mémoire".

Dans une imaginaire machine virtuelle Java (JVM), un objet Java (une instance d'une classe) peut être représenté en mémoire comme une structure qui sera exactement la même pour toutes les instances d'une même classe. Les propriétés (attributs, méthodes, …) sont soit des types primitifs (int, float, …) ou des pointeurs (vers un array, une fonction, …). Cette structure ne contient pas vraiment toutes les données de l'objet, mais plutôt des références, les adresses où sont stockées les données dans la mémoire. Une autre JVM pourrait faire ça différemment : un objet serait représenté par 3 simples pointeurs : le 1er vers un objet "classe" représentant le type de l'objet, le 2^{e} vers une table de pointeurs vers les méthodes de l'objet, le 3^{e} vers la mémoire allouée aux données de l'objet.

Arrivés ici, vous aurez certainement remarqué que je parle principalement de stratégies permettant d'accéder aux propriétés d'un objet qui serait en mémoire. C'est ce qu'on appelle ***property access*** — littéralement "accès à une propriété" — et c'est donc le fait d'accéder au contenu d'une propriété d'un objet. On le fait constamment dans un tas de langages, voici un exemple en JavaScript :

```
1 const o = { // objet
2   f: (x) => parseInt(x, 13),
3   1337: 13.37 + 20
```

```
};

o.f // "property access" ⇒ fonction f
o[1337] // "property access" ⇒ 33.37
o.f(o[1337]) // effectue 2 "property accesses" et un appel de
    fonction ⇒ 42
```

Vous le savez peut-être, JavaScript est un langage orienté-objet basé sur des prototypes (*prototype-based*), pas basé sur des classes (*class-based*). On appelle parfois ces langages *class-free*.

> En JavaScript, les objets sont mutables, on peut donc augmenter les nouvelles instances en leur donnant de nouveaux champs et méthodes. Ces nouvelles instances peuvent à leur tour servir de prototypes pour de nouveaux objets. Nul besoin de classes pour créer des tas d'objets similaires.
>
> Contenu masqué n°1
>
> Source : Douglas Crockford[9]

Non seulement les objets peuvent être créés en "clonant" des prototypes existants, il est également possible d'utiliser la notation *littérale* (aussi nommée "*initializer* notation[18]") pour créer des objets que vous pourrez ensuite tout aussi facilement modifier à la volée.

Créer des objets, les mettre en mémoire, y accéder ou accéder à certaines de leurs propriétés constitue une partie importante de l'exécution de tout code JavaScript. Nous nous concentrons un petit moment sur les objets JavaScript dits "ordinaires" (*ordinary JavaScript objects*[19]) — par opposition aux objets dits "exotiques" (*exotic objects*[20]) — pour évoquer les différentes aspects du *property access*.

2.0.2. Le problème de l'accès aux propriétés : la recherche dynamique est lente !

Et donc, comment implémente-t-on l'accès aux propriétés ? Un bon point de départ serait de regarder ce que la spécification JavaScript actuelle — ECMAScript 2015 — suggère.

Dans la section 9 Ordinary and Exotic Objects Behaviours[21], le point 9.1.8 [[Get]][22] décrit l'algorithme suivant (traduit et simplifié ici pour des raisons, hmm, pédagogiques ?) :

Quand on fait `obj[prop]` ou `obj.prop`, …

1. On s'assure que `typeof prop` est soit `'string'` ou `'symbol'`. (`obj[13]` effectue en réalité `obj['13']`[5].)
2. Si `prop` est une propriété directe de `obj` et que `obj[prop]` n'est pas `undefined`, on retourne `obj[prop]`. Fin.
3. Si `prop` n'est pas une propriété directe de `obj` ou si `obj[prop]` est `undefined`, alors
   - a. Soit *parent* le prototype de `obj`
   - b. Recommençons au point `2.` mais en utilisant *parent* au lieu de `obj`.
   - c. Si *parent* est `null`, on retourne `undefined`. Fin.

— d. On descend la chaine de prototype : retour au point `1.` mais avec *parent* au lieu de `obj` (ce qui signifie qu'on réessaie la même procédure avec `parent[prop]`).

Voici ce qu'on appelle une recherche dynamique, un *dynamic lookup*. On *recherche* une propriété `prop` d'un objet `obj`. C'est *dynamique* parce que lors de l'exécution on cherche `prop` de `obj`, en échouant on recherche `prop` dans le prototype de `obj`, puis `prop` dans le prototype du prototype de `obj`, etc.

On pourrait aussi implémenter un (grand) dictionnaire (ou tableau associatif) où on mettrait tous les objets instanciés par un programme. Les clés seraient des références aux objets et les valeurs seraient à leur tour des dictionnaires avec les propriétés comme clés-valeurs :

```
// Notre JavaScript :
function Point(x, y) {
  this.x = x;
  this.y = y;
}
const p1 = new Point(12, 3);
const p2 = new Point(5, 9);
const p3 = new Point(12);
const lit = { name: 'Bond' };

// Le dictionnaire du moteur JavaScript, stockant les objets :
const allObjects = {
  o0: { // p1
    __proto__: 'p6a1251', // Object{constructor: Point(x, y),
        __proto__: Object{constructor: Object()}}
    x: 12,
    y: 3
  },
  o1: { // p2
    __proto__: 'p6a1251', // Object{constructor: Point(x, y),
        __proto__: Object{constructor: Object()}}
    x: 5,
    y: 9
  },
  o2: { // p3
    __proto__: 'p6a1251', // Object{constructor: Point(x, y),
        __proto__: Object{constructor: Object()}}
    x: 12
  },
  o3: { // lit
    __proto__: 'p419ecc', // Object{constructor: Object()}
    name: 'Bond'
  }
}
```

V8 n'est pas implémenté en JavaScript, mais ça donne une idée basique de comment on pourrait stocker tous les objets utilisés par un programme JavaScript à l'aide d'un dictionnaire. (Ce

dernier serait probablement implémenté sous forme de hash table.)

Maintenant, supposons qu'on veuille accéder à `p1.z`. Ici `p1` est le premier objet créé par notre programme et a donc pour référence `o0`. Tentons de suivre l'algorithme qu'on a trouvé dans la spec ECMAScript :

1. On trouve `o0` dans `allObjects` (*recherche* pour obtenir l'emplacement mémoire de l'objet),
2. On trouve une propriété nommée `"x"` dans `o0` (*dynamic lookup* pour obtenir l'emplacement mémoire de la propriété) et on retourne sa valeur. (Si on avait voulu accéder à `p1.x`, on aurait pu s'arrêter ici et retourner `12`.)
3. Comme `o0` n'a pas de propriété nommée `"x"`, on accède à `o0.__proto__` pour chercher dans cet objet une propriété nommée `"x"`, sinon on accède à `o0.__proto__.__proto__` pour chercher une propriété nommée `"x"`, et on répète ceci jusqu'à la fin de la chaine de prototype (*prototype chain*[10]) jusqu'à ce que `__proto__`[11] soit `null` (ce qui signifie qu'aucun prototype n'a été trouvé).

Vous l'aurez deviné, cette procédure n'est pas efficace et les performances en souffriraient.

2.0.3. La solution à l'accès aux propriétés : les *Maps*

Au lieu de se résoudre à effectuer des recherches dynamiques pour accéder aux propriétés, V8 a implémenté ce concept de *classes cachées* que j'ai mentionné précédemment. À l'origine, ce concept a été inventé pour l'implémentation d'une machine virtuelle pour un autre langage orienté-objet basé sur des prototypes : le langage *SELF*. Voici une citation de l'*abstract* de l'article de 1989 qui publiait pour la première fois cette idée (l'emphase est mienne) :

> […] notre implémentation de SELF s'exécute deux fois plus vite que la plus rapide implémentation de Smalltalk, **malgré, en *SELF*, le manque de classes** et de variables explicites.
> **Pour compenser l'absence de classes, notre système repose sur des *maps*, conçues au niveau de l'implémentation, afin de grouper de façon transparente les objets clônés à partir d'un même prototype** […]
>
> Contenu masqué n°2
>
> Source : C. Chambers, D. Ungar, and E. Lee. "An Efficient Implementation Of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes." *SIGPLAN Not. 24*, no. 10 (September 1989) : 49–70.[12]

Les *classes cachées* sont une dénomination alternative de ce concept de *Maps*. Elles sont à ne pas confondre avec la structure de données nommée *Map* et leur implémentation en JavaScript les *Map Objects*[13]. Même si le nom *hidden class* est très répandu dans la littérature produite par l'équipe de V8, la codebase de V8 y réfère unanimement sous le nom de *Maps*.

Bien que ce concept soit entré en contact avec JavaScript par le biais de V8, la grande majorité des moteurs JavaScript modernes utilisent désormais également cette approche. Safari JavaScriptCore[14] les appelle *structures*. ChakraCore, moteur de Microsoft Edge, les appelle *types*[23]. Dans le moteur SpiderMonkey utilisé par Firefox, elles sont applées *formes* (*shapes*) :

> Il y a un certain nombre de structures de données dans SpiderMonkey qui sont dédiées à rendre rapide l'accès aux propriétés des objets. La plus importante d'entre elles sont les *Formes*. […] Les *formes* sont liées entre elles en une séquence linéaire appelée "lignée[6] de forme", qui décrit les agencements d'un objet. Certaines *lignées de forme* sont partagées et vivent dans des "arbres de propriétés". D'autres lignées de forme ne sont pas partagées et appartiennent à un seul objet JS ; ces derniers étant "en mode dictionnaire".
>
> Contenu masqué n°3
>
> Source : Nicholas Nethercote15 (lire plus16)

(Notez la représentation mentale qu'on peut se faire d'un truc appelé à la fois "*classes* cachées" "*map*", "*structure*", "*forme*", et qui est en lien avec la façon dont sont "*agencées*" les propriétés d'un objet.)

Tout le monde, donc, semble utiliser une variante de ces *Maps*, mais à quoi ressemblent-elles, comment sont-elles générées ? Revenons à V8 et regardons ceci de plus près. Reprenons également notre exemple :

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
const p1 = new Point(13, 37);
const p2 = new Point(2);
const p3 = new Point();
const p4 = new Point(4, 2);
```

De votre lecture ici, vous pourriez vous attendre à ce que le moteur JavaScript créé une *Map* pour `Point` aussitôt que notre premier point est assigné à `p1`, puis réutilise cette ***Map*** pour `p2`, `p3` et `p4`. Pas tout à fait. Nous avons en fait affaire à 3 *Maps* qui ont un lien entre elles mais qui sont différentes. (Ce que j'explique ensuite était bien expliqué dans une page désormais morte que Google titrait "*V8 design reference*". Je vais me permettre de paraphraser un peu ce qu'on y trouvait.)

Commençons par la première partie de notre code. On définit une fonction `Point` qui servira de constructeur pour plusieurs points. Elle prend deux paramètres, `x` et `y`, et ses objets se souviendront des arguments passés à ce constructeur de *Point*s.

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
```

À ce moment, V8 met la fonction en mémoire. Mais ce qui nous intéresse vient ensuite, que se passe-t-il quand nous *créons* un point, un objet `Point` ?

```
1 const p1 = new Point(13, 37);
```

Lorsque V8 constate pour la première fois l'utilisation de la fonction `Point`, ici utilisée comme constructeur, pour créer un nouvel objet basé sur le prototype `Point`, V8 n'a encore aucune idée de ce à quoi ressemblera un `Point`. Tout ce que V8 connait, c'est `function Point`, donc il créé une *Map* initiale `C0` de la *Map* dont `p1` aura besoin.

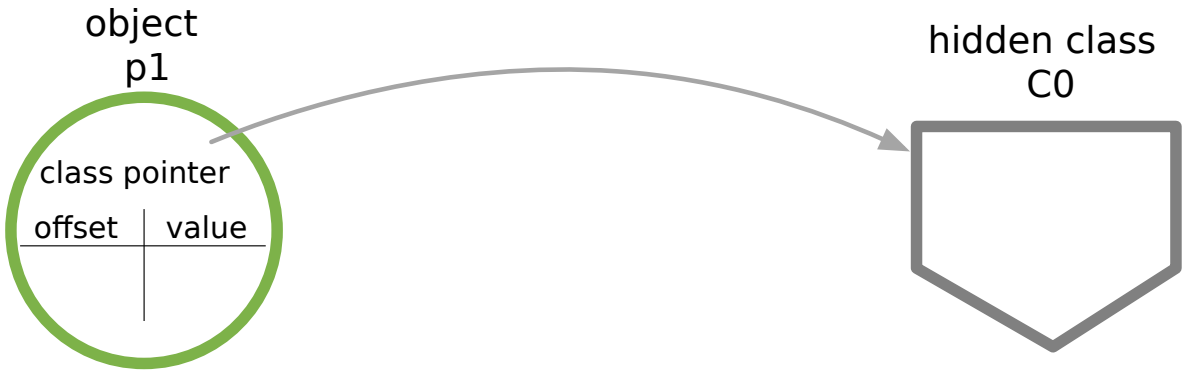

FIGURE 2.

Cette première *Map* `C0` représente un objet `Point` sans aucune propriété (`{}`). Puis V8 créée une variable `p1` sous forme d'un bout de mémoire ne contenant rien d'autre qu'un *pointeur de classe*. `p1` est donc un pointeur vers sa première *Map*, `C0`.

En entrant dans la fonction `Point` avec nos arguments `13` et `37`, la prochaine chose que V8 rencontre est `this.x = x;`, ce qui se résout en `this.x = 13;`. Aha ! Le point `p1` (qui est `this` ici) a une propriété nommée `x` et cette propriété n'est pas référencée par la *Map* `C0` ! Première priorité, V8 met `13` en mémoire et l'adresse de ce `13` est enregistrée dans le premier emplacement mémoire de cet objet (une sorte d'armoire à tiroirs réservé aux *données* d'un objet, où chaque tiroir pointe vers une propriété). Cet emplacement, on va le désigner par `offset 0`. V8 crée ensuite une nouvelle *Map* `C1` basée sur `C0`. L'avantage de `C1`, c'est qu'elle possède une référence vers une propriété nommée `x` dont le contenu est présent en mémoire à l'adresse stockée à `offset 0` de l'objet !

Puis V8 modifie `C0` pour que `C0` puisse se rendre utile la prochaine fois qu'on ajouterait une propriété nommée `x` à un objet de *Map* `C0`, en disant à l'objet en question de *transitionner* vers `C1`.

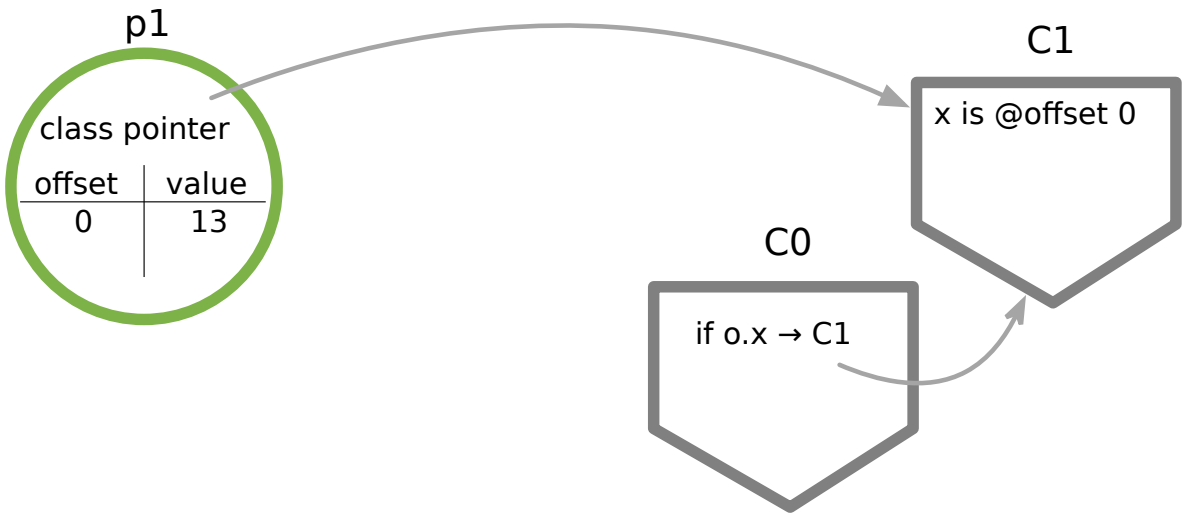

FIGURE 2.

Ligne suivante : `this.y = y;`, donc `this.y = 37`. Voici ce qui se passe sous le capot : D'abord `37` est mis en mémoire et l'adresse placée au prochain *offset* mémoire, `offset 1`. Puis une nouvelle *Map* `C2` est créée en clônant `C1`. Et `C2` se voit attribuer une propriété nommée `y`, dont la valeur pour un objet donné pourra être trouvée à l'adresse `offset 1` de cet objet !

Maintenant que nous avons une *Map* un peu plus utile que `C1`, le *plan de transition* de `C1` est mis à jour pour que tout objet ayant la *Map* `C1` sache qu'il peut adopter la *Map* `C2` s'il recevait une propriété nommée `y`.

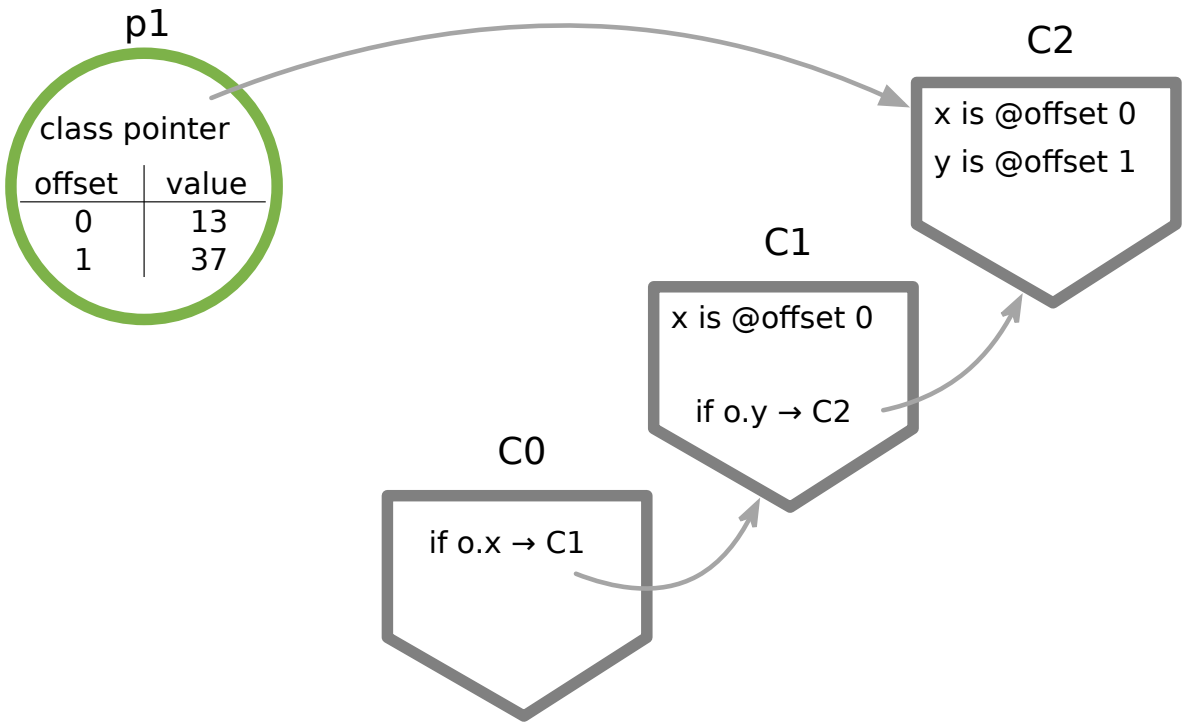

FIGURE 2.

Revenons une dernière fois à notre exemple :

```
const p1 = new Point(13, 37);
const p2 = new Point(2);
const p3 = new Point();
const p4 = new Point(4, 2);
```

J'ai dit qu'il y avait ici 3 *Maps* au total, et j'ai montré par quelles *Maps* passait `p1` avant de se voir finalement assigner `C2`. Vous pouvez très probablement en déduire que ces 4 points finiront par avoir les *Maps* suivantes :

```
/* C2 */ const p1 = new Point(13, 37);
/* C1 */ const p2 = new Point(2);
/* C0 */ const p3 = new Point();
/* C2 */ const p4 = new Point(4, 2);
```

Si vous voulez tester votre compréhension du concept et des *plans de transition*, voici de quoi s'amuser, toujours avec notre fonction `Point`.

```
const p5 = new Point(13, 37);
p5.a = 'a';
const p6 = new Point(13, 37);
p6.b = 'b';
```

Et voici à quoi ça ressemblerait :

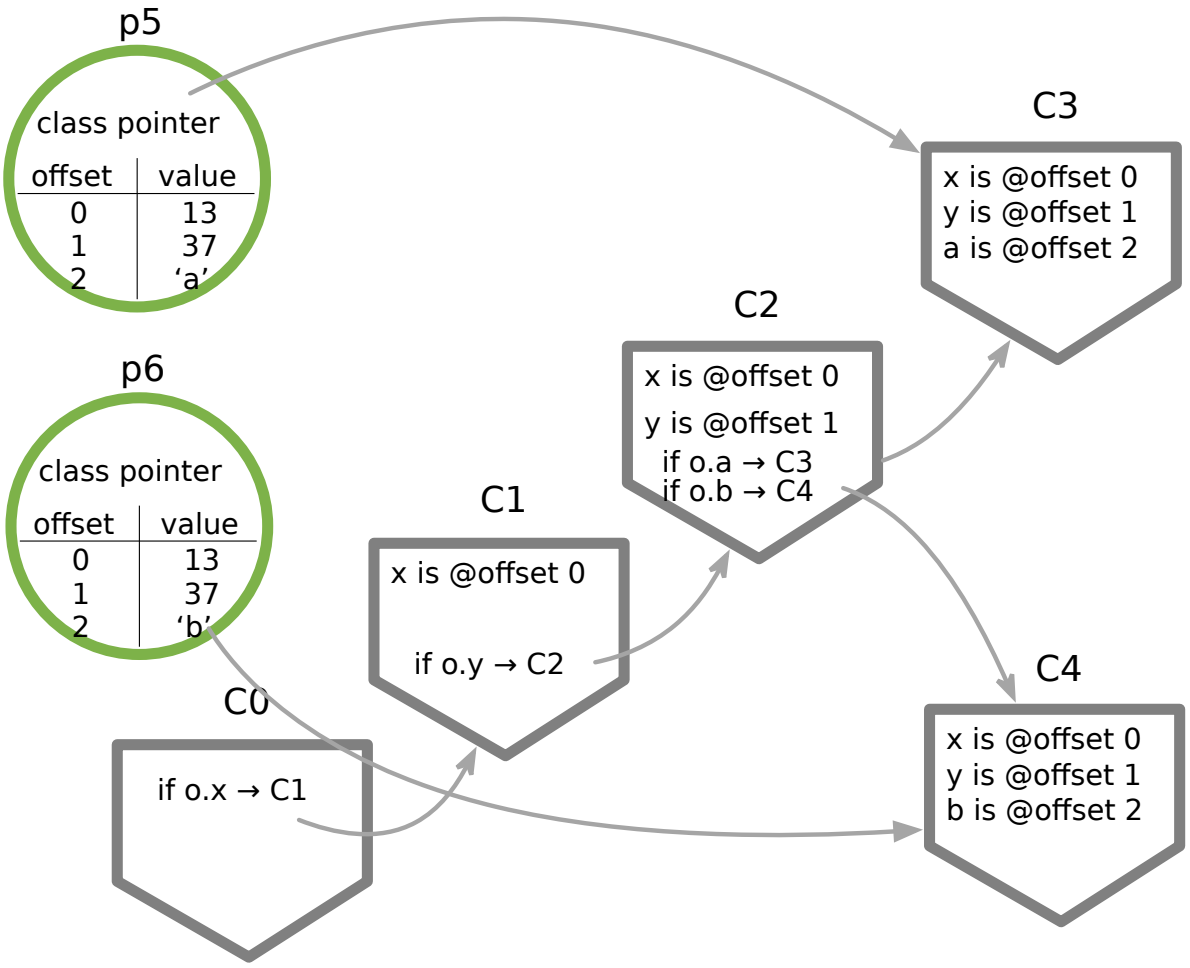

Figure 2.

Une transition n'étant créé que lors de l'ajout d'une propriété à un objet, le plan de transition est un DAG6. Un graphe (dont les noeuds sont des *Maps*) dirigé (il y a des flèches, donc une direction, allant d'un noeud à un autre) acyclique (il n'est pas possible de partir d'un noeud et

d'y revenir, ce serait un cycle).

6

7

8

9

10

11

12

13

14

15

16

17

18

2. J'offre un smoothie à celui qui trouve la super contrepèterie cachée dans cet article. Rendez-vous dans les commentaires. Prenez ça pour un concours. Un prix Concours , même.

3. Lars Bak a passé les 30 dernières années à implémeter et optimiser des machines virtuelles. Il a travaillé sur SELF, Strongtalk, HotSpot, V8, et la VM Dart. Les meilleures parties de V8 proviennent de son expérience passée. De SELF (similaire à JavaScript dans le sens où tous deux sont OO orientés prototypes), V8 a hérité des *Maps*, de l'*inline caching*, l'*inlining* et les désoptimisations. Les conclusions des travaux sur Strongtalk sont devenues une part importante du succès d'HotSpot. Qui par la suite a influencé V8 (JIT). Puis Dart est arrivé, s'inspirant à son tour de Smalltalk, JavaScript, C# et Erlang, sa VM ne gardant que le meilleur de SELF, Strongtalk et HotSpot. D'ailleurs ce nom, *HotSpot*, vous avez remarqué ? Il vient directement de sa capacité à profiler l'exécution du bytecode pour cibler les efforts d'optimisation de la VM sur les "hot spots" - les parties du code qui sont le plus fréquemment utilisées. Tout comme V8 le fait17.

4. Excuse-moi de te dire ça, mon pauvre José, mais tu confonds un peu tout. Tu fais un amalgame entre le mot-clé et son concept. Tu es fou.

5. Si vous ignoriez que `xs[0]` est en réalité `xs['0']`, vous pensiez probablement que JavaScript avait un vrai concept d'*Array*s. Ce n'est pas le cas, et c'est très bien expliqué par James Mickens dans ce talk : https ://vimeo.com/111122950 , regardez de la minute 5 :50 à 12 :20 si vous êtes pressé. Mais vous raterez le début, et ça c'est dommage. Et vous manquerez la suite, et c'est peut-être encore plus dommage.

6. Le terme "lignée" est ici utilisé dans le sens de descendance, héréditaire, voire patrilinéaire24 si les gros mots sont permis. Comme quand on parle de la lignée d'un caniche (ou des rois de France).

6. https ://en.wikipedia.org/wiki/Directed_acyclic_graph
7. https ://draft.li/blog/2016/01/15/one-simple-trick-for-javascript-performance-optimization/
8. https ://www.youtube.com/watch ?v=hWhMKalEicY
9. http ://javascript.crockford.com/prototypal.html
10. https ://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain
11. https ://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/Proto
12. http ://dl.acm.org/citation.cfm ?doid=74878.74884
13. http ://www.ecma-international.org/ecma-262/6.0/#sec-map-objects
14. https ://en.wikipedia.org/wiki/WebKit#JavaScriptCore
15. https ://blog.mozilla.org/nnethercote/2011/11/01/spidermonkey-is-on-a-diet/
16. http ://www.masonchang.com/blog/2008/5/28/spidermonkeys-secret-object-sauce.html
17. https ://draft.li/blog/2016/01/22/chromium-chrome-v8-crankshaft-bailout-reasons/#Crankshaft_and_bailouts
18. https ://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer

19

20

21

22

23

24

3. En pratique

Que se passe-t-il si on **supprime** une propriété d'un objet ?

```
const p = new Point(1, 2);
delete p.y;
p.y = 10;
```

Imaginons que la suppression d'une propriété ajoute une transition disant "si la propriété nommée `y` d'un objet de *Map* `C2` n'est pas là, on transitionne cet objet vers la *Map* `C5`". On aurait donc un objet de *Map* `C5`, et que se passerait-il si on ajoutait à cet objet une propriété `y` ? Il faudrait qu'il subisse une transition pour revenir à `C2`. Par contre pour arriver à cette conclusion, V8 devra chercher si une *Map* existe pour ce cas, et la trouver le cas échéant. Cette opération serait assez couteuse. On aurait dans nos transitions `C2 sans y -> C5`, `C5 avec y -> C2`. Un cycle. Voilà qui complique considérablement les choses.

Souvenez-vous, j'expliquais que les transitions entre les *Maps* formaient un DAG, donc jamais de cycles. Du coup que se passe-t-il si on supprime une propriété ? V8 décide simplement que l'objet dont on a supprimé des propriétés n'a plus de *Map*, l'accès à ses propriétés ne sera pas rapide. Dans le vocabulaire de V8, on dira que cet objet est en "*dictionary mode*". Il sera traité comme un dictionnaire, de façon assez similaire à l'explication que j'ai donnée en début de chapitre *Le problème de l'accès aux propriétés*, avec le premier exemple de code qui s'y trouve.

Voici une capture d'écran des devtools de Chrome, montrant l'état des trois objets suivants :

```
const p1 = new Point(1, 2);
const p2 = new Point(2, 3);
const p3 = new Point(3, 4);
delete p3.y;
```

19. https ://tc39.github.io/ecma262/#sec-ordinary-object
20. https ://tc39.github.io/ecma262/#sec-exotic-object
21. http ://www.ecma-international.org/ecma-262/6.0/#sec-ordinary-and-exotic-objects-behaviours
22. http ://www.ecma-international.org/ecma-262/6.0/#sec-ordinary-object-internal-methods-and-internal-slots-get-p-receiver
23. http ://abchatra.github.io/Type/
24. http ://www.cnrtl.fr/definition/patrilin%C3%A9aire

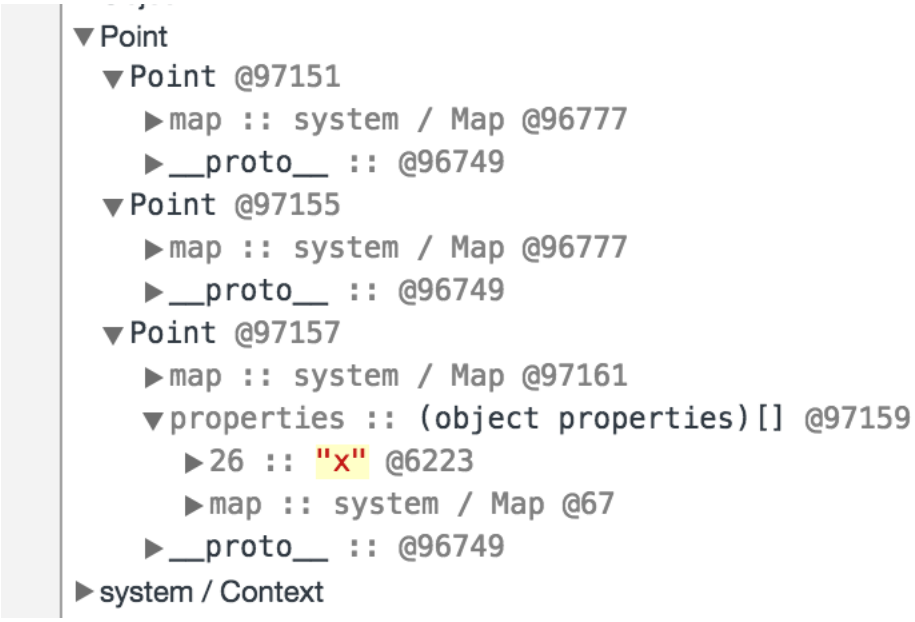

FIGURE 3.

Comme on le constate, les objets `p1` et `p2` ont la même *Map*, la référence à cette *Map* étant `96777`. En revanche le troisième objet, `p3`, a une *hashmap* de propriétés : il est donc en *dictionary mode*.

J'ai condensé quelques éléments importants de cet article dans le benchmark suivant :

```
'use strict';
var _shuffle = require('lodash/shuffle');
var Benchmark = require('benchmark');

function Point(x, y) {
  this.x = x;
  this.y = y;
}

var obj1 = new Point(1, 2);
var obj2 = new Point(2.1, 3.1);
var obj3 = new Point(3, 4);
delete obj3.y;

console.log('obj1 has fast properties:', %HasFastProperties(obj1));
console.log('obj2 has fast properties:', %HasFastProperties(obj2));
console.log('obj3 has fast properties:', %HasFastProperties(obj3));

var len = 1000;

var access = _shuffle('xy'.repeat(500).split(''));

var suite = new Benchmark.Suite();
// cas 1
```

```
suite.add('obj1', function() {
  var xs = [];
  for(var i = 0; i < len; i++){
    xs.push(obj1[access[i]]);
  }
})
// cas 2
.add('obj2', function() {
  var xs = [];
  for(var i = 0; i < len; i++){
    xs.push(obj2[access[i]]);
  }
})
// cas 3
.add('obj3', function() {
  var xs = [];
  for(var i = 0; i < len; i++){
    xs.push(obj3[access[i]]);
  }
})
// cas 1b
.add('obj1b', function() {
  var xs = [];
  for(var i = 0; i < len; i++){
    xs.push(obj1[access[i]]);
  }
  eval();
})
// cas 2b
.add('obj2b', function() {
  var xs = [];
  for(var i = 0; i < len; i++){
    xs.push(obj2[access[i]]);
  }
  eval();
})
// cas 3b
.add('obj3b', function() {
  var xs = [];
  for(var i = 0; i < len; i++){
    xs.push(obj3[access[i]]);
  }
  eval();
})
.on('cycle', function(event) {
  console.log(String(event.target));
})
.run({ 'async': false });
```

À noter :

— Avant le `delete`, `obj1`, `obj2` et `obj3` ont la même *Map*.
— `%HasFastProperties` est une fonction "native" de V8. Pour que cette syntaxe soit prise en compte, il faut passer le *flag* `--allow-natives-syntax` à node ou à Chrome.
— Les cas `1`, `2` et `3` passeront par l'*optimizing compiler*.
— Les cas `1b`, `2b` et `3b` ne passeront pas par l'*optimizing compiler*, ce dernier se désistant sans même tenter d'optimiser le code à cause de la présence d'`eval`. La *bailout reason* sera d'ailleurs *Function calls eval*.
— (Exécuter ce code requiert `npm install benchmark lodash`.)

Résultats :

```
$ node --allow-natives-syntax access-props.js
obj1 has fast properties: true
obj2 has fast properties: true
obj3 has fast properties: false
obj1  x 52,314 ops/sec ±0.82% (91 runs sampled)
obj2  x 53,153 ops/sec ±1.02% (85 runs sampled)
obj3  x 16,956 ops/sec ±0.89% (90 runs sampled)
obj1b x 22,948 ops/sec ±0.78% (89 runs sampled)
obj2b x 23,495 ops/sec ±1.09% (89 runs sampled)
obj3b x 10,632 ops/sec ±1.03% (89 runs sampled)
```

Ce qu'on constate :

— Il n'est pas possible d'avoir un *accès rapide aux propriétés* d'`obj3` car on a supprimé des propriétés de cet objet. Cela a rendu cette fonction 3.3x plus lente.
— Le *bailout* causé par `eval` dans les 3 derniers cas rend l'exécution de ces fonctions 2.3x plus lente. Environ 2x plus lent, c'est pas énorme. Mais c'est parce que ces fonctions sont triviales. Il n'y a pas grand chose à y optimiser. Dans d'autres cas, un *bailout* rendra le code 10, 20 ou 50x plus lent.

S'il fallait conclure, et si vous deviez retenir autre chose que ce qu'est un *bailout* et comment fonctionnent les *Maps*, je vous dirais de rester attentifs aux choses suivantes :

— Si une fonction est plus lente que prévu ou critique en terme de performances, vérifiez qu'elle ne subit de pas *deopt* ni de *bailout*. Par exemple, n'utilisez pas `eval` ou `try/catch` dans ces fonctions. Si nécessaire, pour `try/catch`, isolez ces blocs dans de nouvelles fonctions, ou isolez les blocs qui ont besoin de performances dans des fonctions sans `try/catch`.
— N'utilisez jamais `delete`.
— Pour éviter de polluer la RAM avec des centaines de *Maps*, initialisez toutes les propriétés possibles d'un objet : mieux vaut avoir un objet dont les propriétés sont initialement nulles (`null`) que d'en ajouter constamment à la volée. Si vous devez vraiment en ajouter à la volée, faites-le toujours dans le même ordre (`o.x =` puis `o.y =`).

Cet article n'explique que quelques éléments de fonctionnement d'un moteur JavaScript. Il parle suffisamment de JavaScript pour intéresser les développeurs JavaScript, et ne parle que peu

de développement JavaScript de sorte qu'il puisse rester une lecture intéressante pour d'autres développeurs. J'espère qu'il vous a plu, c'est un sujet dont on parle rarement.

Il y a bien sûr d'autres concepts très intéressants dans nos moteurs JavaScript. Je pense notamment aux `IC`, les *Inline Caches*, sujet sur lequel beaucoup se méprennent. Cela pourrait constituer le sujet d'un article futur.

Il est également à noter que V8 va lentement migrer vers une infrastructure de compilation JavaScript un peu différente de celle de Crankshaft, nommée Turbofan, où pour la première fois un interpréteur sera présent. Il y aura d'abord une compilation initiale JS -> Bytecode, effectuée par Ignition, puis le bytecode sera interprété au moins une fois, puis aura lieu une compilation Bytecode -> Langage machine pour le *hot code*. C'est un changement très intéressant qui s'opère ici, et ses raisons en sont tout aussi intéressantes !

Si vous voulez détecter ces célèbres *bailouts*, j'ai créé/rassemblé quelques outils sous forme d'un site web où vous pouvez analyser du code JavaScript, c'est disponible sur mono.morph.ist . Observer les *Maps* de vos objets dans leur habitat naturel est parfaitement faisable à l'aide des *devtools* de Chrome.

Je me réjouis de vous retrouver dans les commentaires ou sur les forums, merci de votre lecture. Grand merci aux relecteurs, correcteurs et validateurs qui ont pris le temps de relire, corriger ou valider. En particulier, merci à elyppire, Kje, GCodeur, Anto59290, Saroupille, Taguan et Arius pour leur relecture attentive.

Contenu masqué

Contenu masqué n°1

Objects are mutable in JavaScript, so we can augment the new instances, giving them new fields and methods. These can then act as prototypes for even newer objects. We don't need classes to make lots of similar objects. Retourner au texte.

Contenu masqué n°2

[…] SELF implementation runs twice as fast as the fastest Smalltalk implementation, **despite SELF's lack of classes** and explicit variables.

To compensate for the absence of classes, our system uses implementation-level *maps* to transparently group objects cloned from the same prototype […] Retourner au texte.

Contenu masqué n°3

There are a number of data structures within SpiderMonkey dedicated to making object property accesses fast. The most important of these are Shapes. […] Shapes are linked into linear sequences called “shape lineages”, which describe object layouts. Some shape lineages are shared and live in “property trees”. Other shape lineages are unshared and belong to a single JS object ; these are “in dictionary mode”. Retourner au texte.

Liste des abréviations

DAG Directed Acyclic Graph. 13, 15

deopt deoptimization. 3, 18

JIT Just In Time compiler. 2, 3, 14

OO Orienté Objet. 4, 14