Université de Montréal

Rétro ingénierie des modèles d'objets dynamiques pour JavaScript

par
Dalil Boudraa

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître en sciences (M.Sc.)
en informatique

Mai, 2013

RÉSUMÉ

La compréhension des objets dans les programmes orientés objet est une tâche importante à la compréhension du code. JavaScript (JS) est un langage orienté-objet dynamique, et son dynamisme rend la compréhension du code source très difficile. Dans ce mémoire, nous nous intéressons à l'analyse des objets pour les programmes JS. Notre approche construit de façon automatique un graphe d'objets inspiré du diagramme de classes d'UML à partir d'une exécution concrète d'un programme JS. Le graphe résultant montre la structure des objets ainsi que les interactions entre eux.

Notre approche utilise une transformation du code source afin de produire cette information au cours de l'exécution. Cette transformation permet de recueillir de l'information complète au sujet des objets crées ainsi que d'intercepter toutes les modifications de ces objets. À partir de cette information, nous appliquons plusieurs abstractions qui visent à produire une représentation des objets plus compacte et intuitive. Cette approche est implémentée dans l'outil JSTI.

Afin d'évaluer l'utilité de l'approche, nous avons mesuré sa performance ainsi que le degré de réduction dû aux abstractions. Nous avons utilisé les dix programmes de référence de V8 pour cette comparaison. Les résultats montrent que JSTI est assez efficace pour être utilisé en pratique, avec un ralentissement moyen de 14x. De plus, pour 9 des 10 programmes, les graphes sont suffisamment compacts pour être visualisés. Nous avons aussi validé l'approche de façon qualitative en inspectant manuellement les graphes générés. Ces graphes correspondent généralement très bien au résultat attendu.

Mots clés: Analyse de programmes, analyse dynamique, JavaScript, profilage.

ABSTRACT

Understanding of objects in object-oriented programs is an important task for understanding the code. JavaScript (JS) is a dynamic object-oriented language, Its dynamic nature makes understanding its source code very difficult. In this thesis, we focus on object analysis in JS programs to automatically produce a graph of objects inspired by UML class diagrams from an execution trace. The resulting graph shows the structure of the objects as well as their interconnections.

Our approach uses a source-to-source transformation of the original code in order to collect information at runtime. This transformation makes it possible to collect complete information with respect to object creations and property updates. From this information, we perform multiple abstractions that aim to generate a more compact and intuitive representation of objects. This approach has been implemented in the JSTI prototype.

To evaluate our approach, we measured its performance and the graph compression achieved by our abstractions. We used the ten V8 benchmarks for this purpose. Results show that JSTI is efficient enough to be used in practice, with an average slowdown of around 14x. Moreover, for 9 out of the 10 studied programs, the generated object graphs are sufficiently small to be visualized directly by developers. We have also performed a qualitative validation of the approach by manually inspecting the generated graphs. We have found that the graphs generated by JSTI generally correspond very closely to the expected results.

Keywords: Program analysis, dynamic analysis, JavaScript, profiling.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES

JS	JavaScript
JSTI	*JavaScriptTypeInference*
JSVM	JavaScript Virtual Machine
SDR	Structure de Données Récursive

REMERCIEMENTS

Avant tout, je tiens à remercier mon directeur de recherche, Dr Bruno Dufour, qui m'a tant soutenu, m'a apporté son appui et m'a donné tous les outils nécessaires pour mener à bien ma maitrise.

J'aimerais ensuite adresser des remerciements particuliers à Dr Marc Feeley, pour ses conseils enrichissants tout au long de ce travail. Je voudrais exprimer ma reconnaissance envers mon tuteur, Dr Houari Sahraoui.

Aussi, je remercie également les membres du jury, Dr Aboulhamid, El Mostapha et Dr Philippe Langlais d'avoir accepté la charge d'évaluer le travail de ce mémoire.

Je présente ma gratitude à l'équipe du laboratoire GÉODES, du laboratoire DLT, et à toutes les personnes de mon département qui ont participé, de loin ou de près, à la réalisation de ce travail.

Enfin, j'aimerais dédier ce travail à la mémoire de mon cher père, décédé durant ma première session de maitrise, et je remercie ma douce et adorable femme Safaa, qui a fait preuve de patience. Durant lesquels, j'étais souvent absent pour me consacrer à la réalisation de mon travail de recherche. À mon rayon de soleil, ma chère petite fille Zaineb, qui a récemment vu la lumière.

CHAPITRE 1

INTRODUCTION

JavaScript (JS) est un langage de programmation orienté objet, développé en 1995 par Brendan Eich à Netscape. JS est un langage objet de script facile à utiliser, conçu pour créer des applications pour les navigateurs web [20]. Il permet aux non programmeurs d'étendre les sites web contenant du code exécutable, et d'ajouter de l'interactivité aux pages web. JS est devenu un langage très populaire, notamment avec l'avènement d'applications telles que Gmail, Google Docs et Facebook. De nos jours, la grande majorité des sites web utilisent JS. La popularité croissante de bibliothèques telles que JQuery, Prototype et Mootools, a permis d'augmenter de façon significative la taille et la complexité des applications web développées.

Malgré le fait que JS a été initialement créé pour l'interactivité du web, il est de plus en plus utilisé hors des navigateurs web. Par exemple, Sun labs utilise JS comme un langage de programmation des systèmes pour développer un environnement dynamique de programmation appelé Lively kernel [19]. Node.js est une plateforme événementielle pour écrire des applications réseau en JS [29]. CoffeScript est un langage de programmation compilé vers JS ; son compilateur est lui-même écrit en CoffeeScript [9].

1.1 Introduction à JS

JS est fondamentalement basé sur les objets. Seuls quelques types primitifs sont supportés (booléens, nombre à virgule flottante, chaîne de caractères). Tous les autres types de valeurs sont représentés sous forme d'objets, incluant les tableaux et les fonctions. Les objets sont des collections de paires nom-valeur. Les noms sont des chaînes, et les valeurs peuvent être de n'importe quel type. Par exemple, les méthodes d'un objet sont simplement représentées comme des propriétés de type fonction. Comme JS est un langage typé dynamiquement, le type de valeur associé à une propriété peut changer au cours de l'exécution. Il est aussi très important de noter que les objets peuvent être mo-

difiés après leur création. Des propriétés peuvent être supprimées ou ajoutées aux objets.

JS ne dispose pas d'une notion formelle de classe. Les constructeurs ne sont que des fonctions appelées par l'opérateur `new`. Les fonctions sont d'ordre supérieur et peuvent être anonymes. L'héritage est basé sur la notion de prototype, comme dans le langage Self, où chaque objet créé maintient automatiquement un lien implicite à son prototype, c'est-a-dire l'objet à partir duquel il est considéré dérivé. Pour résoudre la recherche d'une propriété d'un objet lors de l'exécution, l'objet lui-même est consulté en premier lieu. Si la propriété recherchée n'y est pas trouvée, l'objet consulte son prototype, si la propriété recherchée n'y est pas encore trouvée, l'objet consulte transitivement ses liens prototype, jusqu'à ce que la propriété soit trouvée ou la racine est atteinte sans succès. En JS un objet peut changer son prototype lors de l'exécution.

La Figure 1.1 présente un exemple de code JS qui illustre le dynamisme des objets. Cet exemple crée deux objets à partir du même constructeur `point`, mais qui possèdent des structures différentes selon le nombre d'arguments passés au constructeur. L'objet `point_2d` contient deux propriétés `x` et `y` au moment de la création, tandis que `point_3d` contient trois propriétés `x`, `y` et `z`. Donc, la structure des objets créés ne dépend pas seulement du constructeur, mais du contexte où le constructeur est appelé. Le contexte dans cet exemple est défini par le nombre d'arguments passés au constructeur.

L'objet `point_2d` peut aussi être modifié après sa création, par exemple ici l'ajout de la propriété `color`. Généralement une propriété pourrait ne pas être ajoutée pour tous les chemins d'exécution (un chemin d'exécution est la séquence d'instructions exécutées par programme durant une exécution donnée). La structure des objets dépendrait alors du chemin et de l'environnement d'une exécution.

1.2 Problématique et motivation

La compréhension de programmes est une partie essentielle de la réutilisation, la maintenance, l'inspection, la rétro-ingénierie, la réingénierie, la refactorisation de code, et d'autres activités dans le génie logiciel. Une fraction importante du temps de maintenance est souvent consacrée à la lecture du code de façon à comprendre la fonctionnalité

```
var point = function (x, y, z){
      this.x = x;
      this.y = y;
      if (arguments.length > 2)
            this.z = z;
};
var point_2d = new point(3, 2);
var point_3d = new point(3, 2, 4);
...
point_2d.color = 'red';
...
```

Figure 1.1 – Exemple du dynamisme des objets en JS

du programme.

Habituellement, les développeurs ont une certaine structure d'objets et d'héritage à l'esprit lors de l'écriture de leur code. Ils ont généralement défini à l'avance l'ensemble des propriétés de chaque objet et le type de chaque propriété qu'un objet détient. Le dynamisme de JS rend la compréhension du code très difficile. Particulièrement, comprendre la structure des objets est une tâche ardue dans bien des cas. Par exemple, le constructeur est souvent insuffisant pour déterminer les propriétés présentes pour un type d'objet spécifique puisque des propriétés peuvent être ajoutées après la création d'un objet. De plus l'héritage en JS est défini dans le code de façon beaucoup moins explicite que dans d'autres langages. Ce problème est exacerbé par le fait que la hiérarchie peut être modifiée durant l'exécution.

Le code de la Figure 1.2 montre la complexité de JS. Le code déclare trois classes `Figure`, `Shape` et `Triangle` dans les lignes `1`, `2` et `3` successivement. La ligne `5` montre que la classe `Triangle` est une sous classe de la classe `Shape` (chaque objet `Triangle` hérite d'un objet de la classe `Shape`). L'objet `figure` est instancié de la classe `Figure` à la ligne `7`. La propriété `shape` accédée par l'objet `figure` dans la ligne `20` n'est pas défini dans le constructeur `Figure`, mais elle a été ajoutée à l'objet `figure`. Notez que cette propriété peut ne pourrait pas ajoutée si l'ajout est conditionnel. Deux méthodes `draw` existent dans la chaine de prototype de l'objet `shape`,

celle déclarée à la ligne `3` et celle à la ligne `6`. Il est donc difficile de comprendre quelle méthode est appelée à la ligne `30`.

1.3 Objectif

Le principal but de ce travail est de produire automatiquement un graphe inspiré du diagramme de classes d'UML à partir d'un programme JS. Ce graphe donne une représentation visuelle de la structure des objets, de leurs relations et de leurs liens d'héritage. Le graphe contient toutes les informations et les propriétés qui ont été ajoutées à des objets pendant toute l'exécution du programme. Cette représentation utile et intuitive pour un programmeur présente deux défis significatifs. Premièrement, il faut trouver les abstractions intuitives, qui aident à la compréhension de classes et de structures de données créées. Deuxièmement, ces abstractions doivent représenter efficacement le nombre énorme d'objets créés durant l'exécution.

1.4 Approche et contributions

La contribution principale de ce mémoire est la conception et l'implémentation de notre outil nommé JSTI (*JS Type inférence*). JSTI est capable de produire le graphe de structure des objets. Ce graphe contient les propriétés, les méthodes, les constructeurs des objets créés, ainsi que l'arbre de la hiérarchie d'héritage. Afin de limiter la taille du graphe, les objets ne sont pas représentés individuellement, mais plutôt à l'aide d'abstractions.

Pour analyser les structures des objets et les visualiser sous forme d'un graphe, nous proposons une approche qui utilise la rétro-ingénierie grâce à l'analyse dynamique. Cette approche rassemble deux contributions spécifiques :

– Un profileur a été développé à fin de recueillir toutes les données nécessaires à la construction du graphe de la hiérarchie des structures des objets et de l'appartenance. Le profileur s'exécute en deux modes : en-ligne et hors-ligne. Dans le mode en-ligne, le profileur et les abstractions s'exécutent d'une façon concurrente avec le code source, et le graphe de types est construit complètement dès la fin

```
1 function Figure(){};

2 function Shape(){};
3 Shape.prototype.draw = function(){};

4 function Triangle(){};
5 Triangle.prototype = new Shape();

6 Triangle.prototype.draw = function(){};
7 figure = new Figure();
...
20 figure.shape = new Triangle();
...
30 figure.shape.draw();
```

Figure 1.2 – Illustration de la complexité de JS

de l'exécution. Pour le mode hors-ligne, le profileur génère une représentation du tas (*heap*) cumulatif à l'exécution avant d'effectuer l'abstraction. Le tas comprend tous les objets créés et les relations entre eux. L'identificateur des structures des objets analyse dans ce cas le tas généré.

– Deux stratégies d'abstraction des objets sont implémentées. La première stratégie est basée sur la notion de la structure des objets seulement. La deuxième est basée sur la notion de la structure des objets et de l'appartenance, qui est inférée à partir des relations entre les objets. Le groupement se base aussi sur la notion de la dominance d'appartenance (*dominance ownership*).

On évalue l'approche sur les programmes de référence de V8. Les résultats obtenus semblent conformes à notre intuition dans la plupart des cas. De plus, nous évaluons l'efficacité des abstractions à réduire la taille des graphes. Les expériences montrent que pour 9 des 10 programmes évalués, la taille du graphe résultant est raisonnable.

1.5 Plan du mémoire

Le reste de ce mémoire est organisé comme suit. Le deuxième chapitre présente un survol des principaux travaux en relation avec notre travail. Le troisième chapitre

explique en détail notre contribution et l'approche proposée. Les résultats et les évaluations de performance de quelques programmes de tests (*benchmarks*) sont présentés et discutés dans le quatrième chapitre. Enfin, le cinquième chapitre est consacré à la récapitulation des principales idées introduites dans ce document et présente quelques perspectives d'améliorations futures de notre travail.

CHAPITRE 2

ÉTAT DE L'ART

Notre travail vise à inférer les structures des objets en JS à l'aide d'une analyse dynamique. De nombreux autres travaux visent à aider les développeurs à mieux comprendre leur code, et particulièrement son utilisation des objets, ainsi qu'à découvrir des informations utiles mais difficiles à extraire directement à partir du code source. Par exemple, plusieurs travaux se sont concentrés sur l'inférence de types pour les langages dynamiques, la rétro-ingénierie du graphe de la hiérarchie des objets pour les langages à prototypes, ou la détection de fuites de la mémoire pour les langages orientés objet tels que Java. Il existe un vaste éventail de travaux existants qui sont reliés à notre approche. Nous catégorisons les travaux les plus directement reliés à notre approche de la façon suivante :

- le profilage de l'utilisation des objets
- l'analyse de la forme
- l'inférence de types

Ce chapitre présente un survol des travaux les plus importants dans chacune des catégories, suivi d'une synthèse des travaux présentés.

2.1 Profilage et instrumentation du code source

Le profilage (*profiling*) est une technique qui s'intéresse à l'étude du comportement du logiciel par l'analyse dynamique des programmes. Le profilage vise à collecter ou analyser des données provenant de l'exécution d'un programme. Par exemple, certains profileurs utilisent les appels à des fonctions pour construire le graphe d'appel, ou analysent l'utilisation des objets pour déduire la structure des données. Le profilage du code est une technique très importante utilisée par les compilateurs à la volée (*just-in-time compilers*) afin d'identifier les parties de code les plus profitables à optimiser [19] au cours de l'exécution.

Rayside et al. [36] introduisent le profilage de l'appartenance des objets (*object ownership profiling*) pour détecter et corriger les fuites de mémoire causées par des références à des objets qui ne sont plus nécessaires et qui les empêchent donc d'être récupérés par le ramasse-miettes (*garbage collector*). Ces objets sont appelés objets indésirables (*junk objects*). Rayside et al. identifient les objets non essentiels par l'observation de leur interaction avec les autres objets dans le programme. La technique profile à partir d'un programme en exécution : les tailles des objets, les moments d'accès à chaque objet, création, collection, accès aux propriétés et aux méthodes de l'objet. Ensuite la technique infère le graphe concret de la hiérarchie de l'appartenance (*inferred ownership hierarchy*), puis elle fait des abstractions des nœuds (objets) qui ont le même : parent (*owner*), type et *overlapping active times*, pour construire un graphe abstrait de l'appartenance où le nœud représente un contexte d'appartenance (*ownership context*). Un objet est observé comme undesirable (*junk*) dans le graphe abstrait s'il est périmé (*stale*), c'est à dire s'il n'a aucune interaction avec les objets essentiels ou avec la racine du graphe. Comme notre approche, le travail de Rayside et al. est basé sur le profilage des objets suivi d'une phase d'abstraction, mais ils ne s'intéressent qu'aux objets essentiels. Par contre, notre approche s'intéresse à tous les objets et leur structure, dans le but de permais consacrmettre la compréhension du code.

Raman et al. [35] analysent le comportement dynamique de la mémoire pour les structures de données récursives (*Recursive Data Structure Profiling*) SDR, comme les listes et les arbres. Ils profilent toutes les instances individuelles des SDR, afin de construire les graphes de la forme (*shape graph*), où les nœuds représentent les SDR et les arcs représentent les liens de dépendance entre les SDR. De plus le profileur associe des événements au graphe de la forme construit, tels que l'intervalle de vie pour chaque instance de SDR et la taille de chaque SDR en mémoire. Raman et al. restreignent leur travail sur les SDR, tandis que notre travail fait une analyse plus générale par le profilage des objets.

L'instrumentation du code est une technique souvent utilisée pour faire le profilage. Une instrumentation du code source ou transformation[1] *source-to-source* consiste à mo-

[1] On entend par transformation toute modification du code.

difier le code par ajout de fragments de code directement au code source du programme, sans affecter sa sémantique. L'instrumentation permet d'observer le comportement du programme, de mesurer de propriétés pendant l'exécution et de contrôler l'exécution.

Reis et al. [38] ont construit et évalué *BrowserShield* pour résoudre les problèmes de vulnérabilités. BrowserShield permet d'intercepter et d'empêcher les attaques contre les pages web et déterminer lorsqu'un script va exploiter le navigateur au moment de l'exécution. Le système *BrowserShield* fait la transformation de source à source des pages web à des pages web strictement équivalentes sémantiquement de façon dynamique. L'instrumentation se fait en deux phases : la première instrumente le code HTML. La deuxième instrumente les *scripts* à l'intérieur des pages HTML. Cette dernière se produit au moment où le navigateur va commencer à exécuter le *script* de la page HTML instrumentée en cours. Sachant qu'un document HTML peut être modifié par l'accès à des propriétés des objets JS en lecture et en écriture, par exemple en utilisant la propriété `innerHTML` pour modifier directement la page web, le principe de *BrowserShield* est d'instrumenter tout accès en lecture ou en écriture à des propriétés des objets et d'insérer des appels à ses propres méthodes, de façon à vérifier au moment de l'exécution si cette instruction va modifier le code HTML ou le script. Au niveau de l'approche Reis et al. s'intéressent à l'accès à des propriétés des objets seulement, tandis que nous intéressons à tous les accès à des objets, leurs propriétés et leurs méthodes. Leur technique vise l'amélioration de la sécurité des pages HTML et les scripts intégrés dans ces pages, par contre nous visons l'amélioration de la compréhension des objets et leur utilisation dans le code source.

2.2 Analyse de la forme (*shape analysis*)

On appelle analyse de la forme toute analyse des codes de programmes, qui s'intéresse à la vérification des propriétés des structures de données allouées pendant l'exécution. Par exemple, dans notre travail nous intéressons à l'analyse des objets pour les codes JS. Plusieurs travaux ont proposé des techniques d'analyse de la forme, qui visent à produire une représentation abstraite de certaines structures de données utilisées par un

programme. Les travaux existants peuvent être divisés en deux catégories : les analyses statiques, basées uniquement sur le code, et les analyses dynamiques, qui observent le programme au cours de son exécution.

2.2.1 Analyse statique de la forme

Quelques travaux analysent la forme des programmes statiquement. Ils visent à identifier les SDR et à comprendre la structure du tas des programmes. Rakesh et al. [13] font l'analyse de la forme des programmes C pour trouver une forme approximative des structures de données dynamiques, par exemple un cycle ou un arbre. Un travail similaire est fait par Sagiv et al. [39, 40], mais ce dernier vise les programmes qui utilisent la mise à jour destructive (*destructive updating*).

Chong et Rugina [7] visent aussi les programmes qui utilisent la mise à jour destructive (*destructive updating*), mais dans le but de présenter le *context-sensitive interprocedural heap analysis*. Ils extraient des informations par analyse statique sur la manière dont les programmes accèdent, en lecture ou en écriture, à des régions de tas (*heap regions*) dans les SDR, comme l'accès à une sous-liste dans une liste. Cette analyse utilise des graphes abstraits de la forme (*abstract shape graphs*) enregistrés dans différents points dans le programme. Ces graphes représentent toutes les informations des accès à des régions des SDR. Ces graphes sont utilisés pour savoir quelles régions ont été accédées par une procédure ou instruction, en lecture ou en écriture, et comprendre les effets des procédures et des instructions sur les SDR. Chong et Rugina analysent la forme des programmes statiquement, et pour déterminer le type d'accès, lecture ou écriture, à des SDR, tandis que notre travail analyse la forme dynamiquement, et dans le but d'analyser les objets.

2.2.2 Analyse dynamique de la forme

L'analyse statique de la forme, tels que les structures de données et les objets, nécessite le calcul du flux de données de manière sensible au contexte (*context-senstive*), ce qui la rend très coûteuse. L'analyse dynamique y est donc souvent préférée. Ce der-

nier type d'analyse de la forme se base sur l'inspection dynamique de relations et des références entre objets. L'analyse dynamique peut être effectuée soit par analyse des instantanés du tas (*heap snapshots*), qui représentent l'état de la mémoire dans un point d'exécution du programme, soit en interceptant les événements correspondant aux écritures de propriétés pour ensuite les analyser.

Pheng et Verbrugge [34] visent à trouver la construction, l'évolution et les modifications de toutes les structures de données d'un programme Java telles que les arbres, les graphes orientés acycliques, et les cycles. Leur approche est basée sur une analyse dynamique d'une trace d'exécution complète d'un programme Java profilé. Ils tentent d'obtenir une image précise et claire des activités du tas durant l'exécution du programme grâce à des traces des événements (*field write event traces*) recueillies à l'aide de *Java Virtual Machine Profiler Interface* (JVMPI)[2]. Ces traces d'événements sont ensuite analysées pour reconstruire des instantanés de tas à chaque *n* événement. Les résultats obtenus par l'analyse dynamique des structures de données sont plus précis que celles trouvées par l'analyse statique [13, 39, 40]. Une autre technique similaire a été développée par Flanagan et Freund [12] pour extraire les structures d'objets. Ils appliquent de plus une séquence d'abstractions des tas reconstitués. Leur modèle objet résultat est proche des objets observés dans les tas captés pendant l'exécution du programme. Flanagan et Freund s'intéressent aux structures des objets seulement. En contraste, Rayside et al. [37] proposent une analyse dynamique pour inférer l'appartenance et le partage des objets (*object ownership and sharing*), ils se basent sur la même technique de *field write event traces*. Ils utilisent des matrices pour sauvegarder les résultats, les colonnes et les lignes des matrices représentent les objets, et les valeurs donnent l'information sur l'appartenance et le partage entre les objets. Rayside et al. ne s'intéressent pas à la structure des objets mais seulement au partage qui vient de l'appartenance. Notre travail rassemble les objectifs des deux travaux précédents, il analyse les structures des objets et leurs appartenances.

L'analyse des tas captés dans des points d'exécution de programmes est une deuxième technique utilisée dans l'analyse dynamique de la forme. Le tas représente un graphe

[2]JVMPI est une interface pour le profilage des programmes Java

d'objets correspondant à un point d'exécution du programme. Le graphe d'objets est une structure complexe constituée de nombreux objets reliés entre eux. Ces relations peuvent être des relations d'héritage ou des relations d'appartenance.

Il faut noter que l'analyse de l'appartenance d'objet (*object ownership*) dans le cadre du graphe des objets signifie l'analyse de liens de référence entre les objets, c'est-à-dire quand un objet réfère à un autre objet par l'une de ses propriétés. Les travaux qui font l'analyse de l'appartenance utilisent souvent l'arbre de dominance pour faire des abstractions des graphes d'objets. Le reste de cette section présente quelques travaux qui analysent les tas de graphes d'objets.

Dans le même but de celui de Rayside et al [36], De Pauw et al. [32, 33] ont proposé des analyses dynamiques de la forme et utilisent des instantanés dans leur technique. Ils comparent deux tas des instantanés qui rassemblent tous les objets et leurs références de relations, l'une prise avant d'effectuer une action critique et une prise par la suite. Cette comparaison permet d'identifier les objets temporaires persistent au-delà de cette période, mais ils ont des références qui détiennent à eux à la fin de la période. Cette technique a été utilisée avec succès pour de grands projets. L'inférence des appartenances dans notre travail se base sur la dominance, tandis que Pauw et al. s'intéressent aux références entre les objets qui se basent sur la comparaison des tas.

Un autre travail d'analyse des tas des programmes Java a été proposé par Hill et al. [17, 18]. Ils visent à extraire l'encapsulation des objets, ou un objet peut contenir d'autres objets à l'intérieur, et les objets internes sont accessibles uniquement via l'objet qui les contient. Ils utilisent un graphe d'objets qui représente la relation d'encapsulation, puis ils appliquent la notion de l'arbres de l'appartenance d'objet (*object ownership trees*), sur le graphe d'objets, pour arriver à le visualiser. Notre graphe d'objet construit est similaire de ces graphes d'objets, mais comprend des différences importantes. Notre graphe représente la hiérarchie et les relations entre les objets, de plus il rassemble toutes les informations reliées aux objets. Aussi, notre travail utilise des abstractions, ce qui n'est pas le cas du travail de Hiell et al.

On note que la plupart des travaux qui s'intéressent à analyser les programmes pour faciliter la compréhension de leur comportement ont proposé des abstractions des don-

nées pour clarifier la visualisation du tas. Aftandilia et al. [1] proposent une technique de visualisation et de navigation, sous forme d'un graphe interactif, du tas d'instantanés des programmes Java en exécution. Ils utilisent deux règles d'abstractions. La première fusionne les objets tels que s'il existe une référence de l'objet `o1` vers l'objet `o2`, et `o1` et `o2` sont de même type, alors `o1` et `o2` soit fusionnés. La deuxième fusionne les objets qui sont du même type et ont la même connectivité, tel que si deux objets `o1` et `o2` ont les mêmes prédécesseurs objets et sont de même type, alors fusionne `o1` et `o2`. Une approche similaire a été proposée par Mark et al. [24]. Mark et al. font l'abstraction de plusieurs graphes d'objets captés dans différents points d'exécution, puis ils font l'union de ces graphes abstraits résultats, ensuite ils appliquent la dominance pour réduire le graphe résultant de l'opération d'union de graphes abstraits. Le graphe résultant n'est pas interactif mais donne une vue plus générale sur le comportement du programme, cette approche est très limité et ne peut pas être appliquée qu'à des tas de petites tailles.

Pour les tas de grand taille, Mitchell et al. [28] visent à analyser et visualiser des tas de tailles contenant des dizaines de millions d'objets, et ils utilisent dans leur analyse un ou plusieurs tas instantanés (*heap snapshots*). De plus, ils prennent en considération la compréhension des structures de haut niveau comme XML DOMs. Ils utilisent plusieurs niveaux d'abstractions, basées sur la notion de dominance. Ils commencent par la construction d'un graphe d'appartenance, où un nœud est un arbre de dominance dans le graphe d'objets. Puis ils font des abstractions de ce graphe d'appartenance par le groupement des nœuds qui sont de même type et ont la même appartenance, pour construire un graphe appelé graphe de sections. Finalement ils construisent le graphe de structures de données, où une structure de données représente un ensemble des sections qui forment un composant fortement connecté (*strongly connected components*). Mitchell et al. analysent et visualisent le graphe d'objets comme nous, mais leur travail est consacré aux programmes Java, où les objets sont statiques et leur type et structure sont définis au moment de la déclaration, et ne changent plus durant l'exécution. Dans le cas de JS, les objets sont dynamiques, ce qui nécessite en plus de trouver une définition claire et efficace de type des objets, et de contrôler et de suivre par la suite le changement de ce type durant toute l'exécution. Il faut noter que notre approche cumule les objets et ne prend

pas en considération leur destruction.

Dietl et Müller [11], analysent les appartenances pour un but différent des précédents. Leur but est d'inférer le modificateur d'appartenance (*ownership modifier*), code peut être inséré au code source, pour qu'un objet dominant assure le contrôle de la modification de tous les objets qui les domine. Dietl et Müller construisent une représentation cumulative de graphe d'objets, pour les programmes Java, similaire à la nôtre pour les programmes JS. Ils enregistrent tous les objets qui ont déjà existé en mémoire, toutes les références entre les objets, et les objets modifiés par d'autres objets. Ils se basent sur le concept de dominance, pour inférer l'appartenance d'objets. Ils ne s'intéressent qu'aux propriétés d'appartenance d'objets. Ils ne s'intéressent ni à l'abstraction de graphe d'objets, ni à la visualisation de graphe des objets abstraite comme Mitchell fait, et ni à la structure interne des objets, de plus l'approche est très limitée et ne s'applique qu'à des tas de tailles limités.

2.3 L'inférence de types

L'inférence de types est un mécanisme qui permet à un analyseur, par exemple un compilateur, de construire ou d'identifier automatiquement les types associés à des variables et à des expressions, sans qu'ils ne soient indiqués explicitement dans le code source.

Plusieurs auteurs s'intéressent à analyser les types et différents travaux ont été proposés dans ce domaine. Par exemple, des analyses de programmes consacrées à l'inférence de types on été développées pour des langages typés dynamiquement comme Scheme et Smalltalk [6, 14, 43]. Ces travaux permettent de clarifier les structures de types pour les programmeurs.

Jensen et al. proposent dans [21] une analyse statique pour inférer et analyser les types dans un programme JS par l'utilisation de l'approche classique de l'interprétation abstraite [10]. Leur travail vise à faciliter la compréhension du comportement de programmes par la production des informations sur les types. Ils visent aussi à détecter l'existence et l'absence des erreurs à la volée, spécialement celles qui sont masquées par

la conversion dynamique de types, par exemple l'invocation d'une valeur quelconque comme une fonction, la lecture d'une variable non définie, ainsi que l'accès à une propriété qui a une valeur *null* ou *undefined*. Notre approche comporte des différences majeures avec leur approche. Notre approche ne vise pas à détecter les erreurs, par exemple la création d'un objet grace à un constructeur qui n'existe plus. Jensen et al. font une analyse statique et la nôtre est dynamique. Ils analysent les objets créés grace à des constructeurs seulement, les objets littéraux n'ont pas été pris en considération dans leur analyse. Ils construisent le graphe de la hiérarchie des objets, mais sans la génération de leur structure, et ils ne visualisent pas les tas de grande taille, car ils ne font aucune abstraction dans leur analyse. De plus, ils n'arrivent à analyser que trois *benchmarks* de V8 à cause de la mémoire insuffisante.

Agesen et al. [2] ont développé un algorithme d'inférence de types pour le langage Self. Similaire à JS, Self est un langage orienté objet typé dynamiquement, et basé sur les prototypes. Agesen et al. ont défini le type d'un objet par un ensemble fini d'objets. L'algorithme de base, défini par Palsberg et al. [30], infère le type de chaque expression et méthode dans le programme. Il associe un ensemble de jetons à chacune d'elles, dans le but de générer les types des objets qui peuvent être évalués par l'expression ou la méthode pour chaque exécution du programme. Donc ils se basent sur des contraintes, et l'ensemble des contraintes, associées aux expressions et méthodes dans le code source, appelées graphe de trace (*trace graph*) des objets. L'algorithme conclut que le programme est correctement typé si les contraintes peuvent être résolues. Cet algorithme est similaire à des travaux antérieurs sur l'inférence de type [4, 5, 15, 16, 22, 23, 25, 27, 41], avec une différence d'utilisation de la dérivation de point fixe pour résoudre les contraintes. L'algorithme de base d'Agesen et al. permet d'inférer les types pour des langages qui supportent l'héritage dynamique, l'héritage multiple, et l'héritage par prototype. Palsberg dans un travail différent [31], propose un autre système de type par contraintes, mais consacré aux objets construits par clonage et concaténation seulement.

D'autre part Anderson et al. [3] ont aussi proposé un système du type statique qui vise à résoudre les problèmes provenant de l'extensibilité des objets par l'ajout dynamique

des propriétés et la mise à jour des méthodes. Le système permet aux objets d'évoluer d'une façon contrôlable pendant l'exécution. Ils ont défini un algorithme d'inférence de type, par la génération des contraintes pour chaque expression. Si l'algorithme réussit, et les contraintes ont des solutions transformables en des types, alors le programme est typé correctement. Ce travail est différent de celui d'Agesen par le fait qu'il vise le langage Self. De plus, Agesen n'infère pas la structure d'objets.

Récemment plusieurs travaux se sont intéressés à analyser les programmes JS pour des buts différents. En particulier, plusieurs se concentrent sur l'analyse des types et des objets pour détecter les erreurs avant et durant la phase d'exécution. Thiemann et al. [42] ont défini un système de type des programmes JS qui suit l'évolution et les modifications d'un objet, pour identifier les erreurs silencieuses qui résultent du type implicite, par exemple le retour de la valeur *undefined* lors de l'accès à une propriété non définie. Ce système signale aussi la conversion suspecte de type comme l'affectation d'une expression arithmétique à un objet. Cette approche a été développée sur une version restreinte du langage JS. Le travail le plus récent s'intéressant à l'analyse des types en JS a été proposé par Chugh et al. [8]. Ce travail définit un système de type basé sur des prédicats, pour supporter les spécifications suivantes du langage : tester les types au moment de l'exécution, le changement dynamique des objets par l'ajout et la suppression des propriétés, le changement des références de relations, et l'héritage par prototype, pour résoudre les problèmes de la dépendance de types. En comparaison avec ces travaux, notre approche se concentre sur les objets et non pas sur les types de base des expressions et variables du programme, comme le type *boolean* par exemple. D'autre part, Chugh et al. ne visent pas la construction de la hiérarchie des objets créés, donc, ils ne font pas l'agrégation des objets et l'abstraction du tas. De plus, notre approche profile les objets et leur structure par accumulation, on ne prend pas en considération la destruction des objets et la suppression de leurs propriétés durant l'exécution.

2.4 Synthèse

Dans ce chapitre, nous avons fait un survol des principaux travaux liés à notre travail de recherche.

Les travaux [1, 24, 28] ont utilisé des abstractions des objets Java proches de la nôtre pour visualiser le tas. Le travail de Aftandilian et al. [1] diffère des deux autres travaux par son intérêt pour la visualisation interactive du tas pour les programmes Java, tels que la hiérarchie d'objets, l'inspection des objets et le contenu des propriétés. Contrairement à notre approche, ces trois travaux n'ont pas besoin d'inférer les types d'objets, par ce que les types et les objets en Java sont statiques, et l'héritage se base sur les classes, donc les objets n'évoluent plus durant l'exécution. Notre travail est différent de telle sorte les objets en JS sont dynamiques ce qui nous oblige d'inférer les types d'objets par le suivi de leur évolution durant l'exécution. De plus, nous nous intéressons à l'ensemble de tous les objets créés durant l'exécution, du début à la fin, et pour tous les instantanés.

La plupart des travaux cités et qui s'intéressent à l'analyse de types ignorent la structure des objets, à l'exception celui d'Anderson et al. [3], qui prennent en considération l'inférence de ces structures. Ils ont défini la structure d'objet d'une façon très similaire à la nôtre qu'ils sont nommé type des objets. Le type d'un objet est la liste des champs et des méthodes présentes dans l'objet. Dans notre travail, contrairement aux travaux d'Anderson et al., cette liste est cumulative : elle accumule toutes les propriétés et les méthodes ajoutées à l'objet pendant toute sa durée de vie. Il faut notre qu'Anderson et al. visent à contrôler le dynamisme de type des objets à la volée, et tandis que notre approche tente de visualiser la hiérarchie et la structure des objets.

CHAPITRE 3

INFÉRENCE DE MODÈLES D'OBJETS

L'approche proposée dans ce mémoire vise à produire un modèle des différents objets créés et manipulés par une application JS. Elle repose sur l'abstraction d'information recueillie à partir d'une exécution concrète d'un programme JS. La section 3.1 donne un aperçu global de l'approche. La section 3.2 explique l'abstraction effectuée. La section 3.3 décrit comment les programmes JS sont transformés pour recueillir l'information dynamique, et finalement la section 3.4 donne les détails de l'implémentation de l'outil JSTI.

3.1 Schéma global de l'approche

Au cours de l'exécution, les objets peuvent contenir des références vers d'autres objets. À un moment donné d'une exécution, l'ensemble de tous les objets concrets et les liens entre eux forment un *graphe d'objets concrets* (GOC). Ce graphe contient typiquement plusieurs milliers d'objets, et ne peut donc pas être visualisé directement par un développeur désirant obtenir un portrait global d'un application. Notre approche vise donc à construire une représentation abstraite plus compacte du GOC.

La modélisation d'objets consiste à créer une représentation abstraite des objets manipulés par une application. Spécifiquement, notre approche consiste à grouper les *objets concrets*, c'est-à-dire les objets observés durant une exécution, en classes d'équivalences selon différents critères d'abstraction. Chaque classe d'équivalence correspond ainsi à un seul *objet abstrait* qui représente tous les objets concrets qui font partie de sa classe d'équivalence. Le résultat de cette abstraction constitue le *graphe d'objets abstraits* (GOA) d'une exécution.

Le GOA produit par notre approche est inspiré du diagramme de classes d'UML. Les diagrammes de classes d'UML permettent de décrire la structure d'un logiciel de façon claire et compacte, et sont couramment utilisés en pratique. Un diagramme de classes

comprend des nœuds qui représentent les classes d'un système. Différents types d'arcs représentent les différentes relations qui existent entre les classes (héritage, association, etc.). Chaque nœud contient le nom de la classe et une description de la structure interne d'une classe, c'est-à-dire une liste de ses attributs et de ses méthodes.

La figure 3.1 illustre l'organisation de notre approche. La première étape consiste à transformer un programme JS par instrumentation du code source en un programme équivalent mais qui permet d'intercepter toutes les opérations sur les objets au cours de l'exécution, par exemple les créations d'objets et les modifications de propriétés. Le code instrumenté peut alors être jumelé à un profileur et exécuté par une machine virtuelle JS (*JS Virtual Machine*, JSVM) afin de construire le GOC au cours de l'exécution. Le GOC représente le comportement du programme durant toute son exécution plutôt qu'un instantané de l'exécution à un point donné. Il sérialise donc tous les objets qui ont été créés ainsi que l'union de toutes relations qui ont existé entre ces objets, mais il ne prend pas en considération la destruction des objets et la suppression des propriétés. Le GOA peut ensuite être produit en appliquant une abstraction au GOC, et exporté sous forme de graphe qui peut alors être visualisé par un développeur.

3.2 Abstraction

Afin de produire un GOA utile pour la compréhension d'un programme, il est nécessaire de trouver une abstraction qui représente adéquatement l'intention du programmeur d'origine. Habituellement, les développeurs ont une certaine structure d'objets et d'héritage à l'esprit lors de l'écriture de leur code. Ils ont généralement défini à l'avance l'ensemble des propriétés de chaque objet et le type de chaque propriété qu'un objet détient. Le code JS reflète ces invariants lors de son exécution. Notre approche vise donc à générer une abstraction qui capture le mieux possible ces invariants.

Notre abstraction prend la forme d'un GOA où les nœuds représentent les objets abstraits et les arcs représentent les relations entre ces objets. Chaque nœud du GOA représente un ensemble de nœuds du GOC qui ont des propriétés communes. Un GOA possède deux types d'arcs permettant de représenter l'héritage et les associations entre

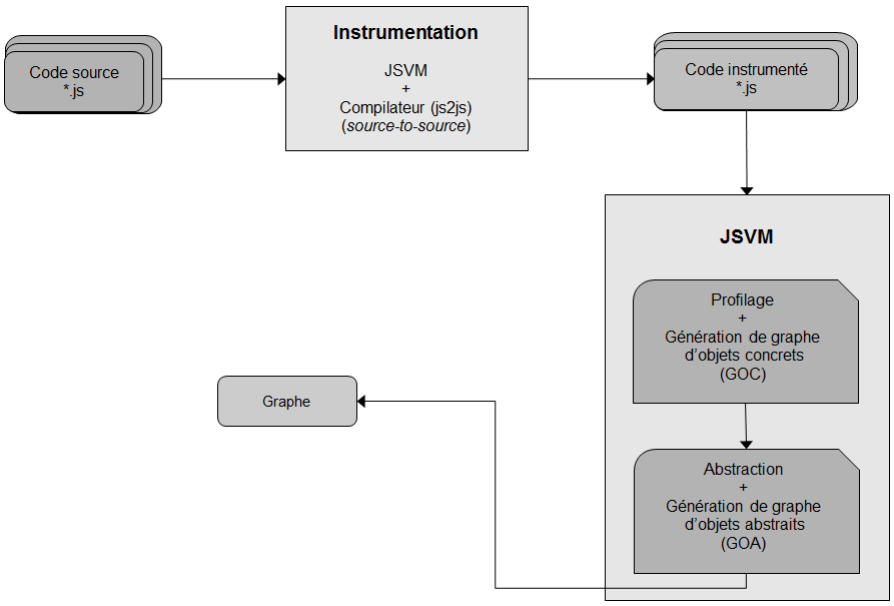


Figure 3.1 – Diagramme de l'approche de génération de GOA

objets. Une association correspond à une référence d'un objet vers un autre. Les association sont représentées dans notre abstraction par un arc étiqueté avec le nom de la propriété concernée : une association d'un objet `x` vers un objet `y` est étiquetée par `p` si `p` est une propriété de `x` qui réfère à `y`. Les arcs non-étiquetés représent les liens d'un objet vers son prototype. Puisqu'un objet peut changer son prototype lors de l'exécution, il est possible pour un objet d'être la source de plusieurs arcs d'héritage dans un GOA.

Le GOA est obtenu par la fusion des nœuds et des associations du GOC selon certaines règles d'abstraction. Définir une abstraction utile et efficace est une tâche difficile en raison de plusieurs particularités du langage JS. Notre approche définit deux stratégies d'abstraction : l'une basée uniquement sur les types des objets, et l'autre qui prend aussi en compte les relations d'appartenance entre les objets (*object ownership*). Par exemple, la figure 3.2 représente un GOC simple contenant trois objets ainsi que leur prototype commun. La figure 3.3 représente le GOA correspondant.

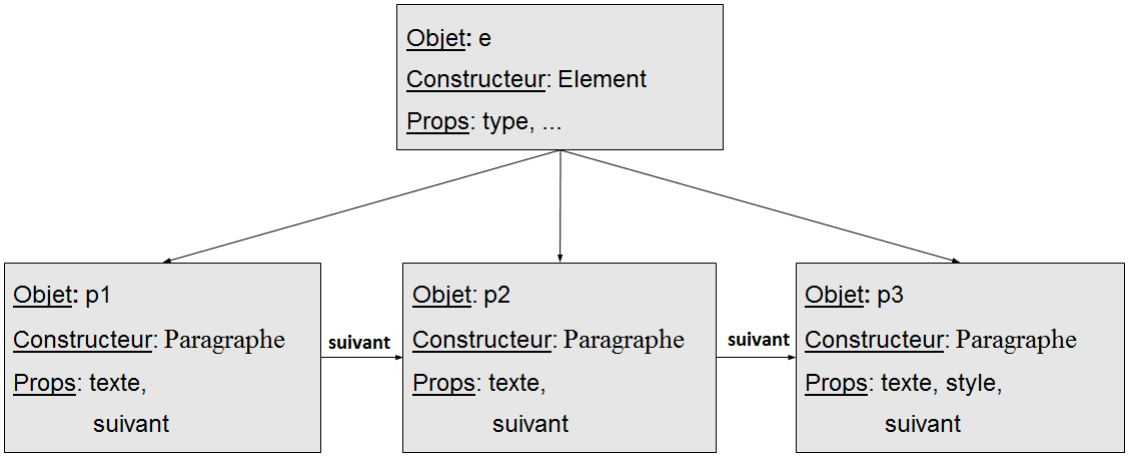


Figure 3.2 – GOC

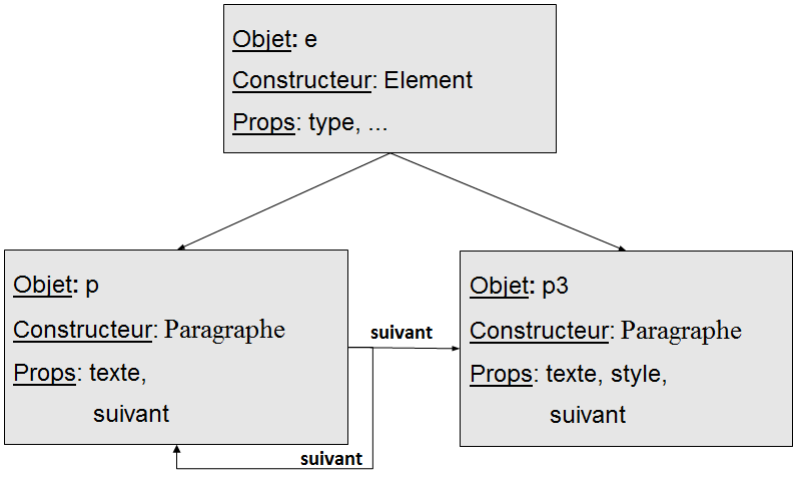


Figure 3.3 – GOA

3.2.1 Abstraction par types

Comme JS est un langage typé dynamiquement, et n'admet donc pas de déclaration de types, il est nécessaire d'inférer des types pour tous les objets. Selon cette stratégie d'abstraction, deux objets concrets sont de même type si ils possèdent les mêmes valeurs pour les attributs suivants :

- **Constructeur :** Si le constructeur est connu, il est défini par son nom et son emplacement dans le code source. Le cas échéant, il est défini par la méthode de création de l'objet.
- **Noms des attributs :** Les noms des propriétés qui possèdent des valeurs de tout type autre que fonction définissent les attributs présents pour un objet donné. Les types et les valeurs de ces propriétés ne sont pas considérés.
- **Noms de méthodes :** Les noms des propriétés qui possèdent des valeurs de type fonction définissent l'ensemble des méthodes qui peuvent être appelées pour un objet donné. Comme dans le cas des attributs, seuls les noms des propriétés sont considérés, et non pas les valeurs qui leur sont associées.
- **Prototype :** L'héritage en JS est basé sur la notion de prototype, comme dans le langage Self. Tous les objets JS à l'exception de l'objet racine possèdent une référence vers un objet prototype duquel ils héritent : lors d'une lecture de propriété, l'objet lui-même est consulté en premier lieu. Si la propriété recherchée n'y est pas trouvée, la chaîne de prototypes est consultée récursivement jusqu'à ce que la propriété soit trouvée ou que la racine soit atteinte sans succès. Il est important de noter qu'un objet peut changer son prototype lors de l'exécution.

3.2.2 Abstraction par appartenance (*ownership*)

En pratique, des objets d'un même type sont souvent utilisés dans des contextes différents. Par exemple, une application peut utiliser plusieurs tableaux pour maintenir des structure de données avec des buts complètement différents. Dans de tels cas, la stratégie d'abstraction par types n'est pas suffisante pour produire une représentation adéquate des objets. Nous proposons donc une stratégie alternative d'abstraction basée sur l'ap-

partenance des objets. Cette stratégie vise à produire une abstraction plus intuitive des structures de données utilisées par une application JS.

Le concept d'appartenance est typiquement défini en fonction de la relation de dominance dans un graphe. Un nœud `d` domine un nœud `n` si tous les chemins depuis la racine `n` passe nécessairement par `d` [1]. Le dominateur immédiat d'un nœud est son dominateur le plus proche, s'il existe au moins un dominateur pour ce nœud. La dominance est souvent appliquée à la visualisation des instantanés du tas (*heap snapshots*). On définit dans ce travail l'ensemble de dominance d'un nœud `d` par l'ensemble de nœuds `s` tel que chaque nœud de l'ensemble `s` est un dominateur du nœud `d`. Le calcul de la dominance est basé uniquement sur les associations (arc étiquetés).

Il est possible de définir une stratégie d'abstraction à partir de la relation de dominance. Deux objets font partie de la même abstraction s'ils possèdent à la fois le même type (tel que défini par l'abstraction précédente) et que leurs dominateurs immédiats respectifs font partie de la même abstraction. Comme cette définition est récursive, les règles d'abstraction sont appliquées au GOC jusqu'à l'obtention d'un point fixe (figure 3.4).

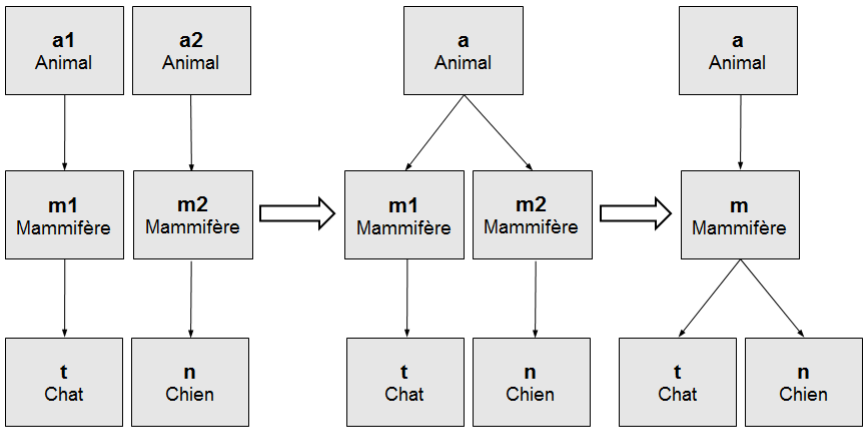


Figure 3.4 – Exemple montre l'application de la récursivité pour générer les appartenances

`Animal`, `Mammifère`, `Chat` et `Chien` sont les types des objets, tandis que `a1`,

`a2`, `a`, `m1`, `m2`, `m`, `t`, `n` dénotent des objets. Le GOC initial (a) contient 6 objets. Après la première itération, le GOA intermédiaire (b) regroupe `a1` et `a2` dans `a` parce qu'ils sont de même types et n'ont pas de dominateurs. Les objets `m1` et `m2` demeurent distincts parce qu'ils n'avaient pas les mêmes dominateurs en (a). La deuxième itération de l'algorithme regroupe `m1` et `m2` dans `m`, qui partagent maintenant le même dominateur.

Les règles de fusion et d'abstraction basées sur l'appartenance utilisées par notre approche sont les suivantes :

– Si les nœud `n1` et `n2` ont le même ensemble de dominance, et `n1` et `n2` sont de même type alors `n1` et `n2` sont fusionnés, tel qu'illustré par la figure 3.5. Dans ce cas, les nœuds `m1` et `m3` ont été fusionnés en un nœud `m` parce qu'ils sont tous deux dominés par le même nœud `a1`.

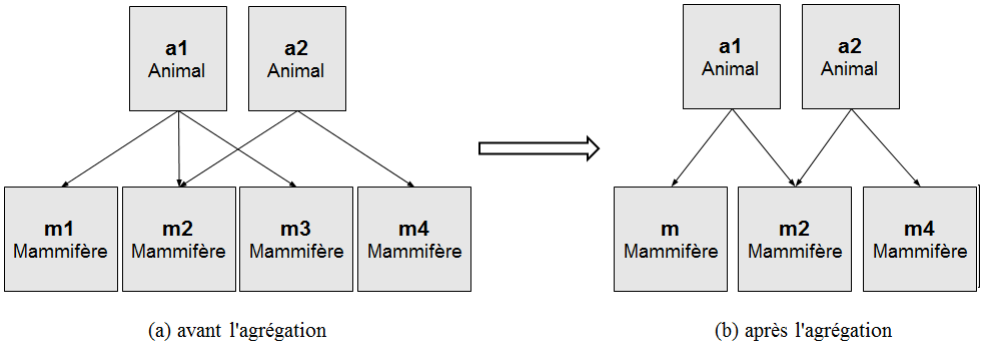
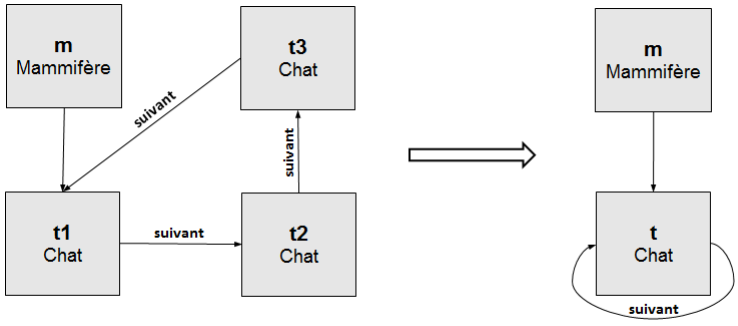


Figure 3.5 – Agrégation basée sur même type - mêmes dominateurs (première règle)

– Si l'ensemble de dominance de `n2` ne contient qu'un seul nœud `n1`, et que `n1` et `n2` sont de même type alors `n1` et `n2` sont fusionnés. Cette règle fusionne les structures récursives de données (par exemple les nœuds d'une liste chaînée), tel qu'illustré par la figure 3.6.

3.2.3 Simplification du GOA

Une pratique courante en JS consiste à implémenter la notion de classe en deux parties distinctes : le prototype associé à un constructeur possède toutes les méthodes alors que les objets concrets créés à partir de se constructeur contienne les attributs. De cette façon, tous les objets héritent les méthodes d'un même prototype, mais leur état peut



(a) avant l'agrégation (b) après l'agrégation

Figure 3.6 – Agrégation basée sur même type - mêmes dominateurs (deuxième règle)

varier indépendamment les uns des autres. Dans un tel cas, les stratégies d'abstraction définies précédemment représentent explicitement cette division. Or, d'un point de vue conceptuel, les objets et leur prototypes correspondent à une seule et même abstraction d'une classe.

Pour résoudre ce problème, notre approche applique une deuxième phase d'abstraction qui permet d'identifier et de regrouper les objets qui se conforment à ce patron. Spécifiquement, si `n1` et `n2` sont deux nœuds d'un GOA, on fusionne `n1` et `n2` si les conditions suivantes sont remplies :

- `n1` est le nœud prototype de `n2` (il existe une association non étiquetée de `n2` vers `n1`)
- La liste de propriétés de `n1` est incluse dans la liste de propriétés de `n2`.
- La liste de méthodes de `n2` est incluse dans la liste de méthodes de `n1`.

3.3 Profilage

La construction des modèles d'objets d'un programme nécessite le profilage des informations sur les objets. Pour cette raison, nous avons développé un profileur qui recueille l'ensemble des objets créés ainsi que leurs attributs et méthodes au cours d'une exécution. Le profileur effectue une transformation du code source de façon à recueillir ces informations et construit une graphe d'objets au cours de l'exécution.

3.3.1 Structures de données de profilage

Au cours de l'exécution, le GOC est construit et maintenu en mémoire. Le GOC accumule toute l'information relative aux objets. Chaque nœud du GOC comprend les informations suivantes :

- `Identifiant` : Identifiant unique généré pour chaque nœud.
- `Parents` : un vecteur contenant les nœuds associés à chacun des prototypes observés de l'objet représenté par ce nœud.
- `Constructeur` : un objet littéral contenant le nom et l'emplacement du constructeur dans le code source de l'application. Cette propriété permet d'identifier de façon même les constructeurs anonymes sans ambigüité.
- `Propriétés` : une structure qui représente l'union de toutes les propriétés ayant été définies pour l'objet représenté. Pour des raisons d'efficacité, une seule table associative est utilisée pour enregistrer à la fois les attributs et les méthodes. Cette structure associe une catégorie à chaque propriété : méthode, référence à un objet, ou autre attribut.
- `Relations` : un vecteur contenant la liste des nœuds référencés par au moins une des propriétés de l'objet représenté.
- `Type` : représente le type de base de l'objet représenté d'après la spécification de JS : `Object`, `String`, `Date`, `Array`, `Boolean`, `RegExp`, ou `Function`.

Lors du début de l'exécution, un graphe initial est initialisé avec un nœud unique qui représente l'objet racine, c'est-à-dire le prototype de tous les objets. Au cours de l'exécution, ce graphe est mis à jour en fonction des créations d'objets et de leur mutations observées.

3.3.2 Instrumentation du code source

Notre approche utilise l'instrumentation du code source pour effectuer l'analyse dynamique nécessaire à la collecte d'information. L'instrumentation consiste à insérer dans le code d'origine des appels aux fonctions du profileur. De cette façon, lorsque le code s'exécute, le profileur est appelé automatiquement lorsque certains événements sur-

viennent. Ceci lui permet notamment d'enregistrer l'information nécessaire en modifiant la structure de données décrite précédemment.

Afin d'observer tous les objets, notre profileur intercepte quatre types d'événements : la création d'un nouvel objet par le code de l'application, l'appel à une fonction, l'écriture d'une propriété d'un objet (ajout ou modification d'une propriété existante) ainsi que la déclaration d'une nouvelle fonction.

3.3.2.1 Création des objets

La construction du GOC nécessite de capter tous les objets créés, ce qui exige d'instrumenter toutes les instructions qui font la création des objets. Nous ne représentons pas explicitement les objets de type `String` dans ce travail car ils ne sont habituellement pas utiles à la compréhension de la hiérarchie des objets en JS.

En JS, on distingue quatre façons de créér des objets : l'opérateur `new`, les accolades `{}` pour créer des objets littéraux (*literal objects*), l'opérateur `[]` qui crée des tableaux, et l'appel à la fonction `Object.create`. La fonction `Object.create(p)` est une fonction de bibliothèque de JS qui crée un nouvel objet avec l'objet prototype spécifié en paramètre.

En plus de ces quatre mécanismes de création d'objets, certaines fonctions de la bibliothèque créent aussi des objets, comme `slice` et `concate` qui s'appliquent à des objets du type `Array`. De plus les développeurs peuvent créer des alias à d'autres fonctions qui permettent la création d'objets, par exemple `myCreate = Object.create`. Dans ce cas, la fonction `myCreate` joue exactement le même rôle de la fonction `Object.create`. Donc, capter tous les objets créés nécessite aussi l'instrumentation de tous les appels à des fonctions.

La stratégie d'instrumentation des créations d'objets est présentée dans le tableau 3.I. Ces instructions sont instrumentées de façon similaire par l'insertion d'un appel à une fonction du profileur. Dans le cas de l'opérateur `new`, une fermeture est insérée pour s'assurer de préserver le comportement du code dans tous les contextes possibles. La fermeture mémorise l'environnement du contexte de création des objets.

Bien que `Object.create` pourrait être traité comme un appel de fonction sans

considération spéciale, l'utilisation de la fonction `recObjptCreate` est utile pour déterminer la méthode de création du nouvel objet.

Les trois fonctions fournies par le profileur sont définies comme suit :

- `recObjAlloc`

  Le rôle de cette fonction est de créer et initialiser un nœud pour le nouvel objet. Ce nœud est alors associé à l'objet par l'ajout d'une propriété cachée `$Model$-node` qui pointe vers son nœud, et finalement ajouté au GOC global. La même procédure est appliquée récursivement au prototype de l'objet s'il n'est pas encore connu. Enfin, les arcs donc cet objet est la source sont ajoutés au GOC.

```
function recObjAlloc(Obj, Constr, Creator, Filename, Position){
    // p = nœud de l'objet prototype de l'objet Obj
    p = abstractObj(Object.getPrototypeOf(Obj));

    // n = nœud de l'objet Obj
    n = abstractObj(Obj);

    // n -> p
    addEdge(n, p);
    addLink(Obj);

    return Obj;
}
```

- `recObjptCreate`

  Cette fonction profile les objets créés par la fonction `Object.create` :

```
function recObjptCreate(Obj, Constr, Creator, Filename, Position){
    // s'assure que la valeur retournée est de type objet
    if (isObject(Obj))
    return recObjAlloc(Obj, Constr, Creator, Filename, Position);

    return Obj;
}
```

  Il est très important de noter que le rôle principal de cette fonction est de sauvegarder la méthode de création de l'objet et de s'assurer que la valeur retournée par

Code d'origine	Code réécrit
`pt_2d = new point(3, 2);`	`pt_2d = (function(){ var temp = point; return recObjAlloc( new temp(3, 2), temp, "new", <ndf>, <pos>); })();`
`pt_2d = {x :3, y :2};`	`pt_2d = recObjAlloc({x :3, y :2}, null, "'{}'", <ndf>, <pos>);`
`pt_2d = [3, 2];`	`pt_2d = recObjAlloc([3, 2], null, "'[]'", <ndf>, <pos>);`
`pt = Object.create(pt_2d);`	`pt = recObjptCreate( (Object["create"])(pt_2d), null, <ndf>, <pos>);`
`point(3, 2);`	`recObjCallExpr(point(3, 2), point, <ndf>, <pos>);`

Tableau 3.I – Instrumentation de la création d'objets

`Object.create` est un objet, car la fonction `create` n'est qu'une méthode définie dans le *prototype* du constructeur prédéfini `Object`. Elle pourrait donc être redéfinie par une application, par exemple :

```
Object.create = function (x){return x;};
o = Object.create(5); // Retourne 5
```

– `recObjCallExpr`

Le profilage des objets retournés par des fonctions comme ceux retournés par la fonction `slice` des tableaux est effectué par la fonction `recObjCallExpr` :

```
function recObjCallExpr(CallExpr, FctName, Filename, Position){
    if (isObject(CallExpr)){
     return recObjAlloc(CallExpr, FctName, "CallExpr", Filename,
                        Position);
    }

    return CallExpr;
}
```

`CallExpr` dénote le résultat de l'appel de la fonction. La fonction `recObjCa-`

`llExpr` teste si ce résultat est un objet. Dans ce cas, elle délègue le traitement de l'objet à la fonction `recObjAlloc`.

3.3.2.2 Écriture de propriétés

Des propriétés et des méthodes peuvent être ajoutées aux objets après leur création. L'ajout peut se faire tout simplement par l'affectation de valeurs à des propriétés non existantes. Le type de valeurs de propriétés peut aussi changer dynamiquement vers un objet. Un objet peut changer son objet *prototype* pendant l'exécution par l'assignation d'un nouvel objet à sa propriété `__proto__`. Ceci rend l'instrumentation des instructions qui font l'accès en écriture et des affectations des valeurs aux propriétés obligatoire. Le tableau 3.II présente l'instrumentation des accès aux propriétés.

Afin de permettre profilage de propriétés et de suivre l'ajout ou le changement de leurs valeurs, nous avons développé deux fonctions : `recSetProp` et `recSetProp__proto__`.

– `recSetProp(Obj, Prop, Val)`

Cette fonction effectue deux tâches :

– Si la propriété n'est pas présente dans l'objet, alors elle est ajoutée au nœud correspondant.

– La propriété est classifiée comme un méthode, une référence ou un autre attribut.

Elle est définie comme suit :

```
function recSetProp(Obj, Prop, Val){
    if (is__proto__(Prop))
        return recSetProp__proto__(Obj, Prop, Val);
```

Code d'origine	**Code réécrit**
`pt_2d.x = 15;`	`recSetProp(pt_2d, "x", 15);`
`pt_2d['x'] = 15;`	`recSetProp(pt_2d, "x", 15);`
`pt_2d[prop_x] = 15;`	`recSetProp(pt_2d, prop_x, 15);`
`pt.__proto__ = pt_2d;`	`recSetProp__proto__(pt, "__proto__", pt_2d);`

Tableau 3.II – Instrumentation des écritures de propriétés

```
    Obj[Prop] = Val;

    ...

    // pour le cas de : x = y["p"] = z; et p est nouvelle dans y.
    return Val;
}
```

– `recSetProp__proto__(Obj, Proto, Val)`

Les objets en JS sont dynamiques et les développeurs peuvent changer le *prototype* des objets par l'affectation à leur propriété `__proto__`. Un objet peut donc avoir plus d'un *prototype* lors de son exécution, mais il ne peut en avoir qu'un seul à la fois. Nous accumulons tous les prototypes d'un objet pendant l'exécution. La fonction `recSetProp__proto__` est responsable de cette tâche :

```
function recSetProp\_\_proto\_\_(Obj, Proto, Val){
    Obj[Prop] = Val;

    if (isObject(Val)){
     ...
    }

    // pour le cas de : x = y["p"] = z; et p est nouvelle dans y.
    return Val;
}
```

3.3.2.3 Déclaration de fonctions

Comme la résolution d'une fonction n'est effectuée qu'au moment d'un appel, notre profileur doit pouvoir associer à chaque objet fonction un emplacement dans le code source pour traiter l'utilisation de cette fonction comme un constructeur. Dans notre approche, cette information est enregistrée comme des propriétés de chaque fonction au moment de sa déclaration. Le tableau 3.III présente la stratégie d'instrumentation des déclarations de fonctions.

La fonction `recFunct` est responsable d'associer à chaque fonction déclarée son nom et son emplacement dans le code source à l'aide de deux propriétés cachées :

```
function recFunct(Funct, Constrname, Filename, Position){
    // ajoute la propriété $name au constructeur déclaré
    Object.defineProperty(Funct, '$name',
        {value: Constrname, writable: true, readable: true,
         enumerable: false});

    // ajoute la propriété $loc au constructeur déclaré
    Object.defineProperty(Funct, '$loc',
        {value: new $Model$_loc(Filename, Position),
         writable: true, readable: true, enumerable: false
        });

    return Funct;
}
```

On distingue deux types de définition de fonction : la déclaration de la fonction dans un contexte local ou global, ainsi que l'affectation d'une expression de type fonction à une variable ou propriété. Ces deux cas requièrent des stratégies d'instrumentation différentes. La déclaration de fonction globale prend effet aussitôt qu'un fichier qui la contient est chargé :

```
foo();
function foo(){
    print("Je suis foo");
}

> js exp.js
> Je suis foo
```

Dans ce code, l'appel à la fonction *foo* précède sa déclaration, mais la déclaration est tout de même visible et l'exécution se termine correctement. Par contre, l'exemple suivant fait appel à une fonction affectée à une variable avant l'opération d'affectation, ce qui résulte en une erreur au moment de l'exécution.

```
var foo;
foo();
foo = function (){
    print("Je suis foo");
}

> js exp.js
foo();
^
>
```

L'enregistrement d'une fonction déclarée localement ou globalement doit donc être effectué lorsque celle-ci devient visible (par exemple, au début d'un fichier pour une déclaration globale).

On distingue dans cette approche deux catégories d'objets :

- **Objets sans constructeur défini :** Le constructeur dans notre approche est considéré `null` pour les objets littéraux (ceux créés par l'opérateur `{}`), les objets `Array` littéraux (ceux créés par l'opérateur `[]`), les objets créés par la fonction `Object.create` et les objets retournés par des fonctions.
- **Objets dont le constructeur est défini :** Les objets sont créés grâce à l'appel à des fonctions prédéfinies dans JS ou à l'aide de l'opérateur `new`. Dans ce cas, la fonction appelée est le constructeur de l'objet créé. Le constructeur est défini par le nom de la fonction et son emplacement dans le code source (le nom du fichier suivi par le numéro de la ligne et le numéro de la colonne). Le tableau 3.IV présente quelques exemples pour un fichier source nommé `exp.js`.

  Comme JS supporte les définitions de fonctions anonymes, il est possible pour une fonction de ne pas avoir de nom. Ceci pose un défi pour la visualisation des résultats. Pour résoudre ce problème, nous nommons les fonctions anonymes selon la technique proposée par Salman et al. [26]. Le tableau 3.V présente un aperçu de cette technique.

Code d'origine	**Code réécrit**

...
function foo(){
 	
...
}
	recFunct(foo, "foo", <ndf>,
	<pos>);
	...
	function foo(){
	var __Tac_This__ = this;
	...
	}

shape =
function Shape(){
 	
...
};
	shape =
	recFunct((function Shape(){
	var __Tac_This__ = this;
	...
	});), "Shape", <ndf>, <pos>));

Tableau 3.III – Exemple d'instrumentation du code de déclaration et d'affectation à des fonctions

Code JS	**Nom du constructeur**	**Location du constructeur**
pt_2d = new point(2, 3);	point	exp.js@1.13
tableau = new Array();	Array	exp.js@4.15
tableau = new Date;	Date	exp.js@12.15

Tableau 3.IV – Exemples de constructeurs

Description	**Code JS**	**Nom fournit**
assigné à une variable	`foo = function (){};`	`foo@1.7`
assigné à une propriété ajoutée	`foo.baz = function (){};` `foo["baz"] = function (){};`	`baz@1.11` `baz@1.14`
propriété d'un objet littéral	`{bar : function(){}};`	`bar@1.10`
indice dans un tableau littéral	`foo = [function (){}];`	`foo@1.3`
indice d'un tableau	`foo[1] = function (){};`	`1@148.1`

assigné à une propriété	`function foo(){`
	`  this.bar = function (){};`
	`};`
		`foo@1.3/bar@1.18`

appelé à l'intérieur d'une fonction	`function foo(){`
	`  function bar(){`
	`  function(){}();`
	`  };`
	`  bar();`
	`};`
		`foo@1.1/bar@1.16`

appelé dans le contexte global	`function(){}()`	`null`
passé comme argument à une fonction	`foo(new function(){}());`	`foo@1.1`

Tableau 3.V – Nommer les fonctions anonymes

3.4 Implémentation

L'outil JSTI implémente l'approche présentée dans ce chapitre. Il permet de construire un GOA automatiquement à partir d'une exécution d'un programme JS. JSTI supporte deux modes d'exécution : hors-ligne et en-ligne. Dans le mode hors-ligne, un GOC est généré lors de l'exécution, et l'abstraction est réalisée après la terminaison de l'exécution. Dans le mode en-ligne, l'abstraction est effectuée lors de l'exécution du programme source sans passer par un GOC.

3.4.1 Mode d'exécution hors-ligne

Dans le mode hors-ligne, le programme profilé s'exécute jusqu'à sa terminaison. Pendant l'exécution, seul le profilage est effectué : aucune abstraction ne s'applique aux objets et un GOC est généré à la fin de l'exécution. Ensuite, le GOC généré est analysé afin de générer le GOA correspondant. La figure 3.7 présente les principales étapes de ce mode d'exécution.

Par souci d'efficacité, le GOC généré par JSTI est un fichier JS exécutable. Ce format permet d'utiliser les opérations optimisées par les machines virtuelles JS modernes. Comme les GOC sont en général très volumineux, la performance de la lecture du fichier est très importante pour rendre l'approche utilisable en pratique.

3.4.2 Mode d'exécution en-ligne

Le mode d'exécution en-ligne diffère de celui du hors-ligne de telle sorte que le profileur et les abstractions s'exécutent d'une façon concurrente avec le code profilé. Le GOA est donc construit complètement dès la fin de l'exécution. La figure 3.8 présente le schéma des étapes d'exécution dans ce mode.

Un avantage important de ce mode est évidemment qui ne requiert pas l'étape coûteuse de génération d'un GOC. Par contre, il est important de noter quelques différences importantes entre les deux modes. Premièrement, comme l'abstraction par appartenance nécessite de connaître au préalable les types des objets, cette stratégie n'est disponible que dans le mode hors-ligne. De plus, même pour les stratégie d'abstraction par types,

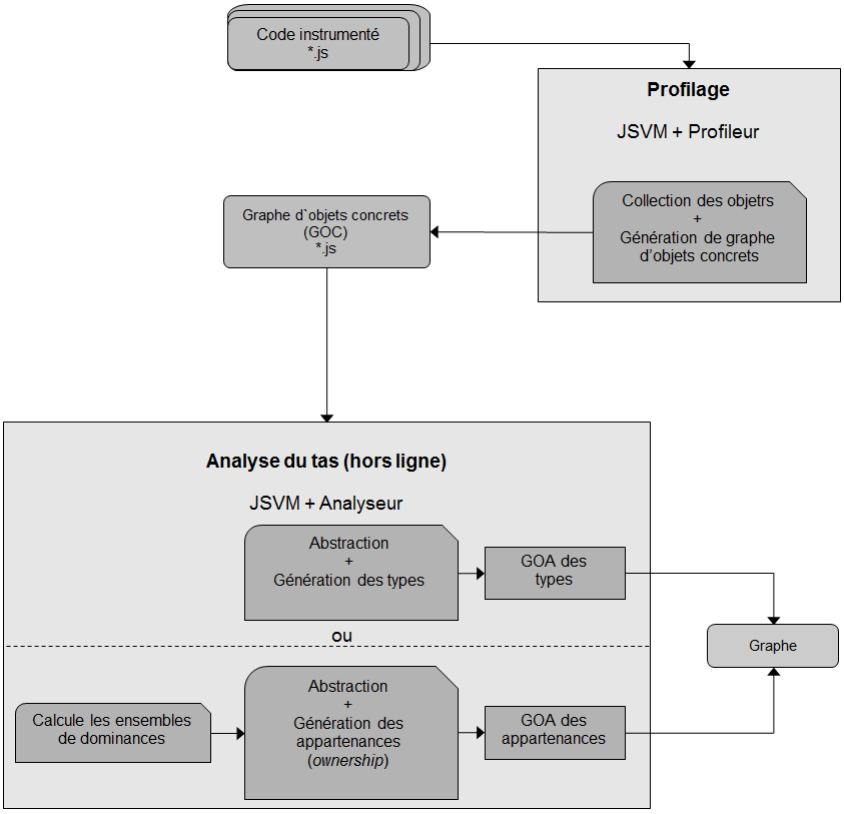


Figure 3.7 – Diagramme de l'approche de génération des types d'objets hors-ligne

les résultats obtenu par l'abstraction en-ligne ne sont pas forcément les mêmes que ceux obtenus par abstraction hors-ligne. Ceci est causé par la nécessité d'effectuer les mises à jour du graphe abstrait en temps réel, ce qui peut causer une perte de précision au cours de l'analyse.

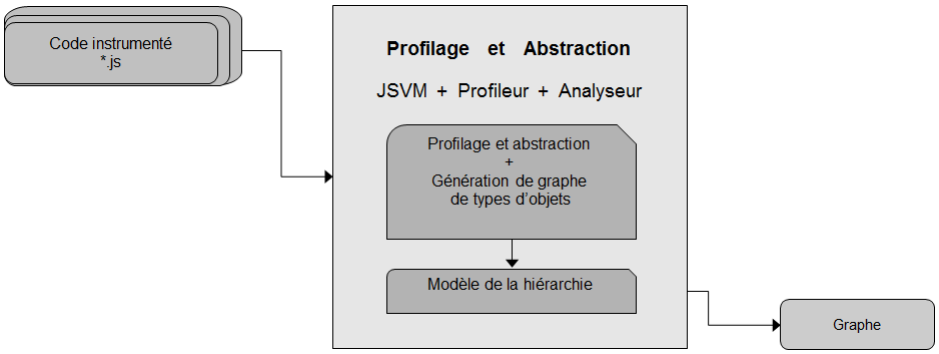


Figure 3.8 – Diagramme de l'approche de génération des types d'objets en-ligne

3.5 Limites

Certains aspects du langage JS ne sont pas supportés par le prototype de l'outil JSTI :

- `with` : L'opérateur `with` permet d'utiliser un objet comme *scope*. Lorsque l'interpréteur rencontre une variable dans un bloc `with`, la recherche est effectuée dans les propriétés de l'objet en premier lieu, puis la recherche se poursuit comme à l'habitude si la propriété n'y est pas trouvée. Notre profileur n'est pas en mesure de détecter les accès aux propriétés effectués dans un bloc `with`.
- `eval` : La fonction `eval` évalue une chaîne de caractères comme du code JS. La chaîne représentant un code JS peut inclure des créations d'objets, accès à des propriétés d'objets existants et déclaration des constructeurs. Intercepter ces événements nécessite d'effectuer de l'instrumentation au cours de l'exécution, ce qui n'est pas supporté par JSTI.
- `Object.defineProperty` : Des propriétés peuvent être ajoutées à des objets en utilisant la fonction `Object.defineProperty`. Cette fonction est rarement utilisée par les développeurs, et a été jugée peu importante à implémenter dans le prototype.

Le support pour ces fonctionnalités pourrait être ajouté à l'outil si nécessaire.

De plus, notre approche augmente de façon importante la consommation mémoire au cours de l'exécution. Elle peut donc causer des problèmes de saturation de mémoire pour la plupart des machines virtuelles JS commerciales puisque celles-ci permettent rarement d'augmenter la taille maximale du tas disponible pour l'application.

CHAPITRE 4

ÉVALUATION

Dans ce chapitre, nous présentons les résultats de l'évaluation de notre approche. Cette évaluation vise deux principaux objectifs. Le premier est de mesurer le ralentissement de l'exécution causé par le profilage et la génération de modèles. Le deuxième est d'étudier la qualité et l'utilité des graphes de modèles d'objets générés. À cette fin, nous avons effectué une étude quantitative qui mesure le facteur de compression entre les graphes d'objets concrets et les graphes des deux modèles définis, ainsi qu'une étude qualitative des graphes générés.

Ce chapitre est organisé comme suit. Nous commençons par définir l'environnement de l'expérimentation ainsi que les programmes de références utilisés et leurs caractéristique. Nous présentons ensuite les métriques utilisées pour l'évaluation quantitative, ainsi que les résultats de la comparaison entre les les deux modes de la génération de modèles de types (en-ligne et hors-ligne). Nous terminons par une étude de cas de quelques programmes de références.

4.1 Environnement d'expérimentation

4.1.1 Configuration du matériel

Nous avons utilisé un serveur de calcul à 4 processeurs six-coeurs *Intel Xeon X5650 2.67GHz*. Chaque processeur possède 12Mb de mémoire cache. La station dispose de 99Gb de mémoire vive et exécute le système d'exploitation Fedora Core 14.

4.1.2 Programmes de références

Pour évaluer notre approche, nous avons effectué des tests sur les programmes de références de V8 :

Crypto : Ce programme de référence écrit par Tom Wu (1698 lignes), fait le cryptage et le décryptage de fichiers textes.

Richards : Richards est un programme de référence écrit par Martin Richards (539 lignes), simule le OS kernel.

Deltablue : Solveur de contraintes, écrit en Smalltalk par John Maloney et Mario Wolczko (880 lignes).

Code-load : Mesure à quelle vitesse un moteur JS peut commencer l'exécution de code après le chargement d'un programme JS, *social widget* étant l'exemple. La source de test est dérivée des bibliothèques *open source* (fermeture, jQuery) (1530 lignes).

NavierStokes : NavierStokes est un solveur des équations 2D, il manipule les matrices. Se basé sur le code de Oliver Hunt (387 lignes).

Box2DWeb : Box2Dweb écrit par Erin Catto, porté à JavaScript. (560 lignes).

Splay : Splay exerce la gestion automatique de la mémoire (394 lignes).

Regexp : Le programme de référence *regular expression* généré par l'extraction des opérations d'expressions régulières (*regular expression*) de 50 pages web les plus populaires (1761 lignes).

Raytrace : Un lanceur de rayons écrit par Adam Burmister (904 lignes).

EarleyBoyer : Est un programme de référence classique de Scheme, traduits en JS par Florian Loitsch par le complicateur Scheme2Js (4684 lignes).

4.2 Étude de l'utilisation des objets

Afin d'obtenir un portrait global de l'utilisation des objets par les programmes de référence, nous avons effectué une étude quantitative de certains aspects de l'utilisation des objets. Cette étude fournit trois types d'informations : le nombre d'objets créés, leur mécanisme de création, et leur dynamisme.

4.2.1 Nombre d'objets créés

Le tableau 4.I affiche le nombre d'objets créés, le nombre de relations de prototype et le nombre d'associations entre les objets, dans le but de montrer les statistiques et de clarifier la relation proportionnelle entre ces nombres.

Pour tous les programmes étudiés, le nombre d'objets dépasse le nombre de relations de prototype de un, car l'objet racine `Object.prototype` possède un prototype `null`. Ceci implique que chaque objet ne possède qu'un seul prototype durant toute l'exécution. Le nombre d'associations est défini par le nombre d'arcs étiquetés dans le GOC et représente le nombre de propriétés qui font référence à d'autres objets. Les programmes de références s'exécutent à l'aide de deux fichiers JS, `run.js` et `base.js`, en plus du code du programme lui-même. L'exécution de ces deux fichiers est responsable de la création de *5* relations de références. Nous observons donc que seulement 7 des 10 programmes contiennent des relations de références. Les programmes *code-load*, *navier-stokes* et *regexp* ne créent que des objets isolés.

4.2.2 Mécanisme de création

Le tableau 4.II montre le nombre d'objets créés selon leur mécanisme de création.

Pour chaque programme, le total des objets créés par les différentes méthodes est inférieur au nombre d'objets créés cités dans le tableau précédent (tableau 4.I), car les objets internes de JS n'ont pas de créateur (par exemple `Array.prototype`).

La plupart des programmes créent la majorité de leurs objets à l'aide de l'opérateur `new`. Les objets littéraux sont rarement utilisés sauf dans le cas de `Splay`, où la majorité de ses objets sont des objets littéraux. Les programmes créent aussi un nombre limité d'objets de type tableau littéral, sauf encore une fois `Splay` où la moitié des objets créés sont des tableaux littéraux. Nous remarquons aussi qu'aucun objet n'est créé par l'appel à la fonction `Object.create`. Seuls les programmes `Deltablue`, `Box2d`, `Regexp` et `Raytrace` qui créent des objets par d'autres mécanismes tels que des appels de fonctions. Dans le cas de `Regexp`, notamment, plus d'un million de nouveaux objets sont le résultat d'appels de fonctions.

4.2.3 Dynamisme

Le tableau 4.III présente le dynamisme des objets dans les programmes JS étudiés. Nous calculons le nombre de changements de types des objets, le nombre de propriétés

Programmes de références	Nombre d'objets	Relations de prototype	Associations
Crypto	6,970	6,969	6,557
Richards	1,359	1,358	2,757
Deltablue	96,010	96,009	111,851
Code-load	3,831	3,830	5
Navier-stokes	98	97	5
Box2d	200,590	200,589	85,381
Splay	1,123,423	1,123,422	1,156,158
Regexp	1,240,836	1,240,835	5
Raytrace	2,197,871	2,197,870	955,259
Earley-boyer	8,816,640	8,816,639	17,618,751

Tableau 4.I – Le nombre d'objets créés par les programmes de référence de V8

Programme de références	L'opérateur 'new'	Objets littéraux '{}'	Tableaux littéraux '[]'	Appel de fonction	Object.create
Crypto	6,951	3	5	0	0
Richards	1,340	2	4	0	0
Deltablue	95,992	2	4	12,852	0
Code-load	3,818	2	4	0	0
Navier-stokes	84	2	4	0	0
Box2d	200,503	26	5	12	0
Splay	44,208	715,682	363,525	0	0
Regexp	55	2	108	1,240,664	0
Raytrace	2,197,739	87	6	33	0
Earley-boyer	8,813,640	5	2,986	0	0

Tableau 4.II – Les méthodes de création des objets dans les programmes de références de V8

allouées aux objets au moment de la création et le nombre des propriétés ajoutées par la suite. Un objet peut changer son modèle de type ou appartenance par l'ajout d'une propriété (ou méthode), par le changement de type de la propriété (par exemple une propriété du type entier qui devient une méthode), et par le changement de son prototype.

Il faut noter que les résultats présentent le nombre de changements de type et non le nombre d'objets qui ont changé de type au cours de l'exécution. Par exemple, si un objet change son type *3* fois et un deuxième objet change son type *5* fois, le nombre de changements de types sera *8* pour ces deux objets.

À l'exception des programmes `Code-load` et `Regexp`, dans le reste des programmes le dynamisme des objets est souvent utilisé. Le nombre d'ajout de propriétés varie de 2 à 736 159 (Raytrace). La malléabilité des objets en JS est donc une fonctionnalité populaire et fortement utilisée en pratique. Par contre, le nombre de propriétés ajoutées aux objets lors de leur création (incluant les propriétés ajoutées par le constructeur) est toujours beaucoup plus élevé que le nombre de propriétés ajoutées dynamiquement.

Nous avons aussi confirmé que le nombre de changements de types (qui n'est pas présenté dans le tableau) est égal au nombre de propriétés ajoutées. Ceci implique qu'aucun attribut ne devient une méthode ou inversement. De plus, il n'y a aucun accès en écriture à la propriété `__proto__`.

Programmes de références	Propriétés déclarées dans les constructeurs	Propriétés ajoutées dynamiquement
Crypto	14,509	8,092
Richards	4,396	131
Deltablue	276,030	201
Code-load	30	2
Navier-stokes	288	20
Box2d	499,107	83,400
Splay	1,971,540	88,315
Regexp	3,629,720	2
Raytrace	6,433,991	736,159
Earley-boyer	17,629,255	4,154

Tableau 4.III – Le dynamisme des objets dans les programmes de référence de V8

4.3 Métriques

Nous utilisons deux métriques pour évaluer l'utilité de notre approche en pratique : le ralentissement de l'exécution et le rapport de compression de graphe d'objets.

4.3.1 Ralentissement

Pour évaluer la performance de notre outil, on mesure le temps écoulé lors de l'exécution du code instrumenté ainsi que le ralentissement de l'exécution causé par le profilage et la génération de graphes de types ou appartenances.

Nous mesurons donc le temps d'exécution de tous les programmes de références analysés pour les différents modes d'exécution et stratégies d'abstraction. Pour chaque cas, nous exécutons les programmes trois fois. Pour chaque programme, la meilleure valeur observée parmi les trois est enregistrée. Notez que le temps d'instrumentation, le temps de la création de GOC et le temps de la génération de graphes ne sont pas inclus.

La façon de calculer le temps de l'exécution pour générer les modèles dépend du mode d'exécution :

- Le mode d'exécution en-ligne : Dans ce mode, le temps d'exécution de génération de types est égal au temps d'exécution du code instrumenté.
- Le mode d'exécution hors-ligne : Dans ce mode, le temps d'exécution de génération de types ou appartenances dans ce mode est la somme de temps d'exécution du code instrumenté et le temps de l'analyse du GOC. Le temps du chargement du GOC n'est pas inclus.

Nous définissons aussi le facteur de ralentissement de l'exécution par la formule ci-dessous. Ce facteur représente le facteur multiplicatif du temps d'exécution instrumenté par rapport au temps d'exécution d'origine.

$$\textit{Facteur de ralentissement} = \frac{\textit{Temps d'exécution du code instrumenté}}{\textit{Temps d'exécution du code original}}$$

Le tableau 4.IV fournit les temps d'exécution mesurés en secondes pour les programmes de V8. Ces résultats montrent les temps d'exécution du code original et du

code instrumenté dans les deux modes d'exécution (les trois premiers colonnes du tableau). Notez que la notation *ND* signifie que le temps d'exécution n'est pas disponible puisque l'outil *JSTI* n'arrive pas à générer le GOA à cause de la saturation de la mémoire. La figure 4.1 affiche aussi les facteurs de ralentissement calculés selon la formule ci-dessus.

Le temps d'exécution de la version instrumentée du code est toujours plus élevé que celui du code original, à l'exception de `Code-load` où moins de *100* objets ont été créés. Le temps d'exécution est n'est donc pas affecté de façon mesurable pour ce programme. Pour les autres programmes, le facteur de ralentissement varie entre 5 et 66 pour la mode en-ligne, et entre 4 et 33 pour la version hors-ligne.

Dans le cas des programmes `Crypto`, `Richards` et `Navier-stokes`, Le tableau 4.IV montre que le temps de génération de types en-ligne est inférieur au temps de génération de GOC. Ces programmes allouent peu d'objets. On peut donc conclure que le coût lié à l'abstraction est plus élevé que le coût associé à la production du GOC. Ceci s'explique par le fait que la construction du GOC est une opération simple en comparaison avec le processus d'abstraction. Une étude plus approfondie du coût des différentes opérations serait par contre nécessaire afin d'obtenir une meilleure compréhension de l'efficacité du mécanisme d'abstraction.

4.3.2 Compression

Pour évaluer l'efficacité des stratégies d'abstraction, nous mesurons le facteur de compression obtenu pour chacune des stratégies en comparaison avec le GOC. Pour chaque GOA, nous pouvons comparer son nombre de nœuds et son nombre d'arcs avec le nombre d'objets et d'arcs présents à l'origine dans le GOC. Nous utilisons les formules suivantes pour cette comparaison :

$$\textit{Facteur de compression de nœuds} = \frac{\textit{Nombre de nœuds de graphe concret}}{\textit{Nombre de nœuds de graphe de modèles}}$$

Programmes de références	Code original	Mode en-ligne	Génération de GOC	Génération de types (hors-ligne)	Génération d'appartenances (hors-ligne)
Crypto	5	102	131	0	85
Richards	2	10	11	0	3
Deltablue	2	10	8	0	1
Code-load	2	2	2	0	0
Navier-stokes	2	55	68	0	0
Box2d	3	58	47	75	236
Splay	2	50	34	35	2,777
Regexp	3	55	39	34	3,428
Raytrace	3	203	88	ND	ND
Earley-boyer	11	454	ND	ND	ND

Tableau 4.IV – Temps d'exécution pour les programmes de références de V8 (en secondes)

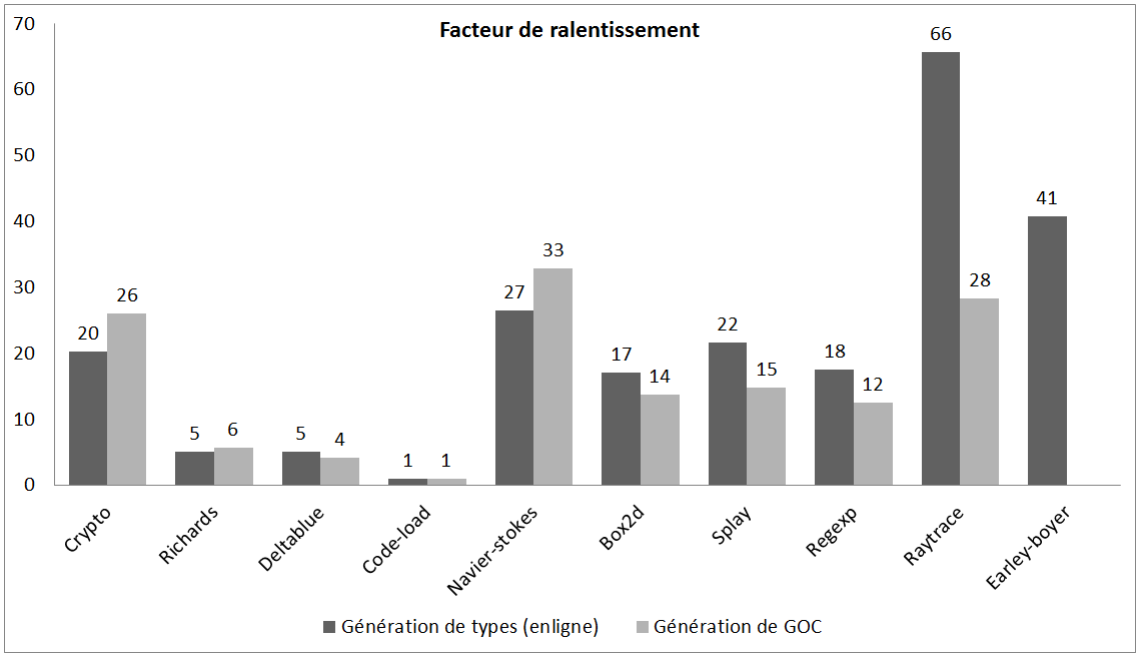


Figure 4.1 – Facteur de ralentissement pour les programmes de références de V8

$$\textit{Facteur de compression d'arcs} = \frac{\textit{Nombre d'arcs de graphe concret}}{\textit{Nombre d'arcs de graphe de modèles}}$$

Les figures 4.2, 4.3, 4.4 montrent les résultats reliés à la compression des graphes pour la stratégie d'abstraction par types et le mode en-ligne.

Les figures 4.5 et 4.6 montrent les résultats pour les stratégies d'abstraction par type et par appartement, respectivement, pour le mode hors-ligne.

Pour 6 des 10 programmes, le modèle abstrait généré par JSTI ne contient pas plus de 30 nœuds. La taille de ces graphes les rend donc faciles à visualiser pour un développeur. Pour *Deltablue*, *Raytrace* et *Earley-boyer*, les graphes abstraits peuvent contenir jusqu'à *60* nœuds, ce qui peut rendre la visualisation difficile. Le programme *Box2d* est un cas particulier : son GOA contient *143* nœuds. La visualisation d'un tel graphe peut être problématique pour un utilisateur.

On remarque aussi que le graphe d'appartenances de *Deltablue* contient *3,907* nœuds, mais après la simplification, le graphe ne contient que *47* nœuds. Ceci démontre l'efficacité de la stratégie de simplification à réduire le nombre de nœuds représentés.

Les graphes de types générés par les deux modes sont différents dans les cas de `Splay` et `Box2d`. La taille de graphe de types généré en mode hors-ligne est plus grande que celle du graphe généré en-ligne. Dans *Splay*, les objets créés par le constructeur `Node@681.1` ont été représentés par le même type (nœud) dans le mode en-ligne tandis qu'ils ont été représentés par *4* types distincts en mode hors-ligne. Le graphe de types hors-ligne est plus précis que celui en-ligne dans ce cas puisque les décisions locales de l'abstraction en-ligne introduisent de l'imprécision dans la représentation des types.

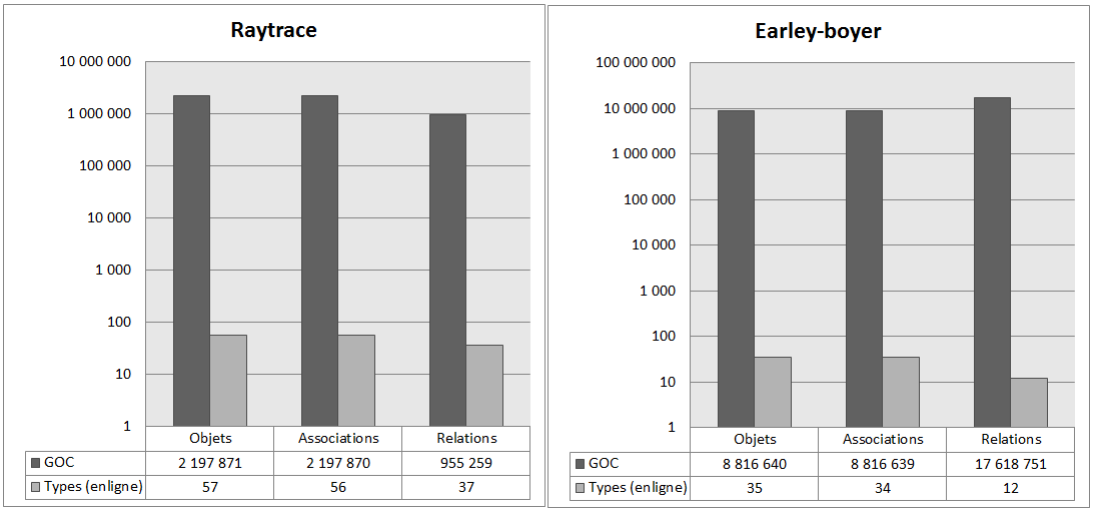


Figure 4.2 – Résultats de compression de graphes d'objets pour les programmes de références de V8

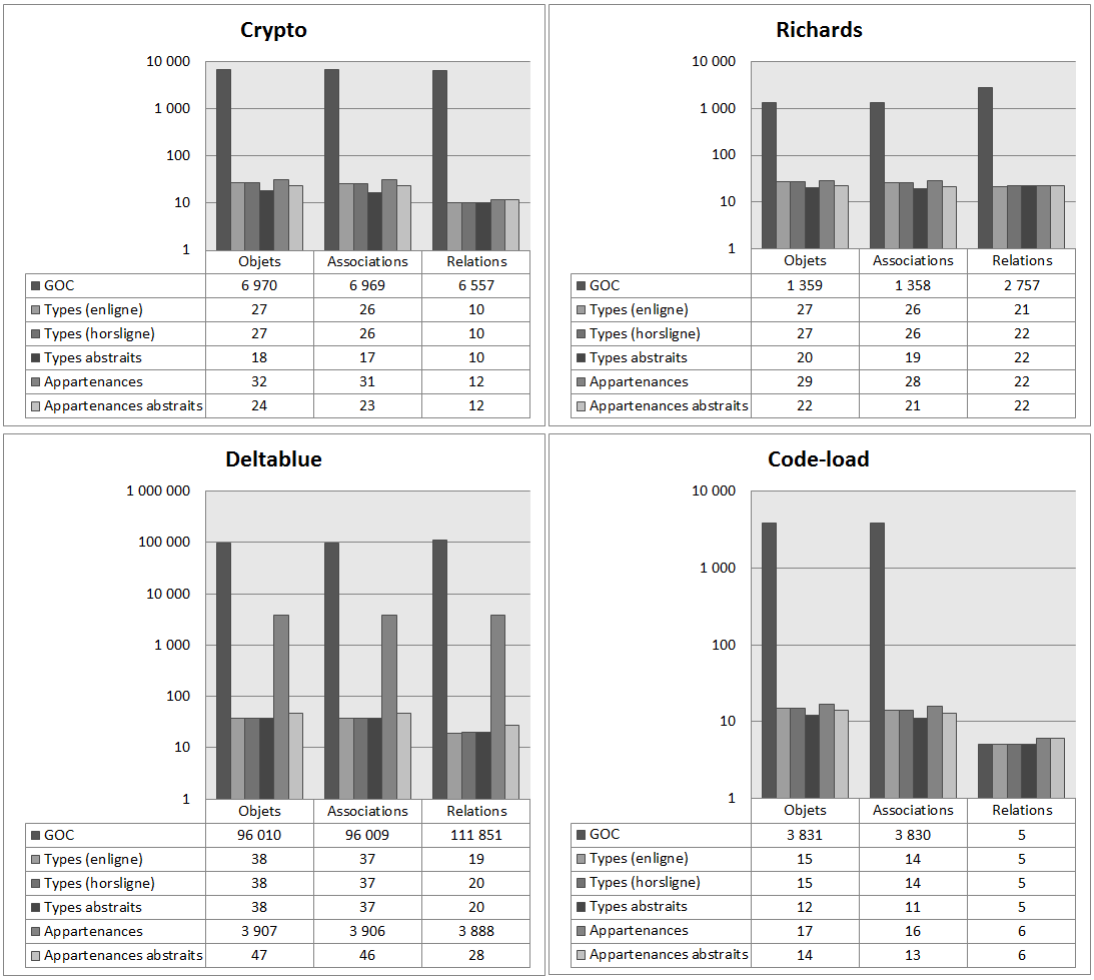
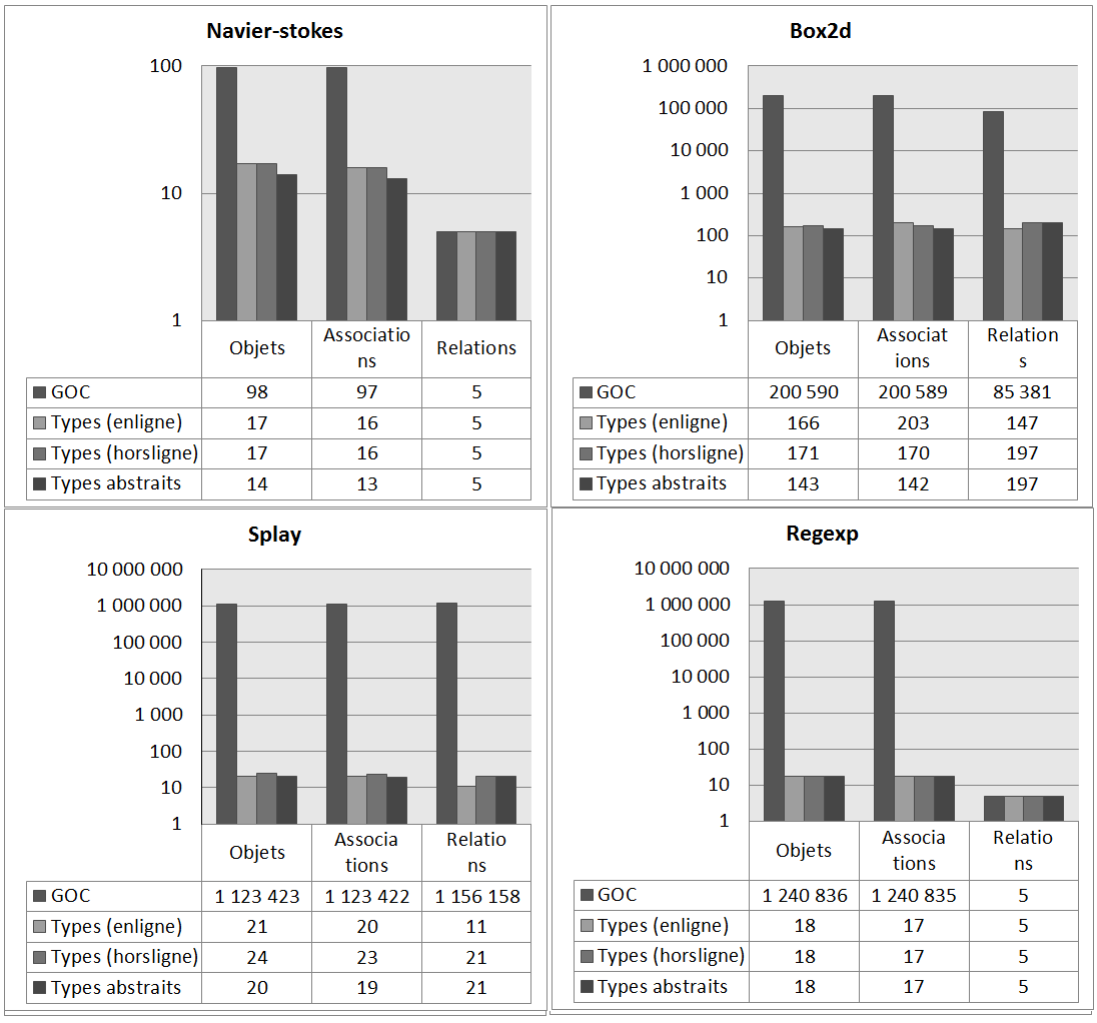


Figure 4.3 – Résultats de compression de graphes d'objets pour les programmes de références de V8



Navier-stokes	Objets	Associations	Relations
GOC	98	97	5
Types (enligne)	17	16	5
Types (horsligne)	17	16	5
Types abstraits	14	13	5

Box2d	Objets	Associations	Relations
GOC	200 590	200 589	85 381
Types (enligne)	166	203	147
Types (horsligne)	171	170	197
Types abstraits	143	142	197

Splay	Objets	Associations	Relations
GOC	1 123 423	1 123 422	1 156 158
Types (enligne)	21	20	11
Types (horsligne)	24	23	21
Types abstraits	20	19	21

Regexp	Objets	Associations	Relations
GOC	1 240 836	1 240 835	5
Types (enligne)	18	17	5
Types (horsligne)	18	17	5
Types abstraits	18	17	5

Figure 4.4 – Résultats de compression de graphes d'objets pour les programmes de références de V8

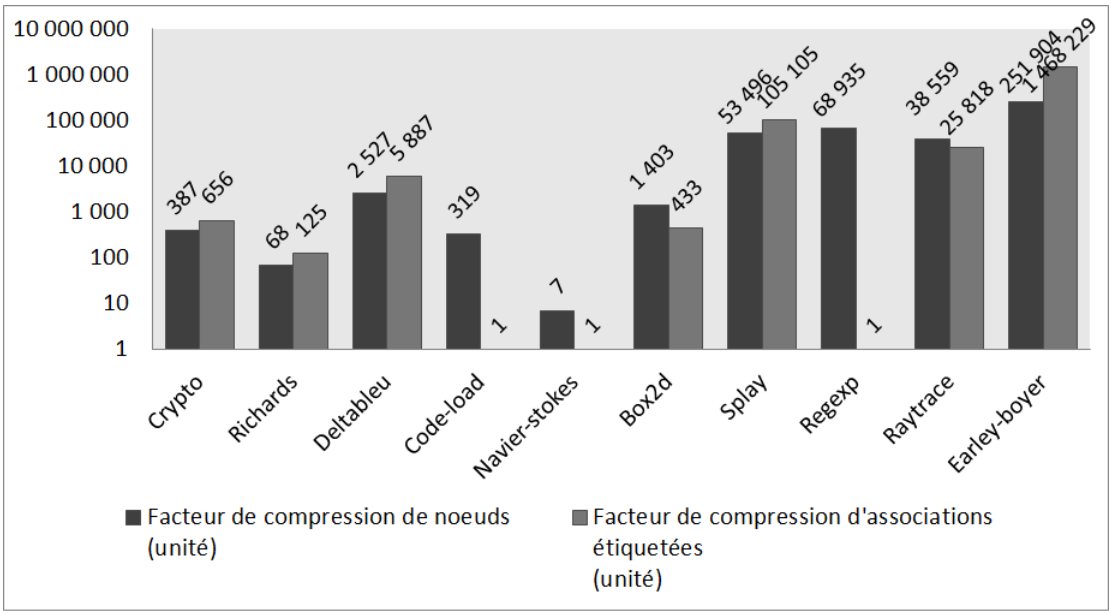


Figure 4.5 – Facteur de compression en comparant le GOC avec le graphe de types abstrait.

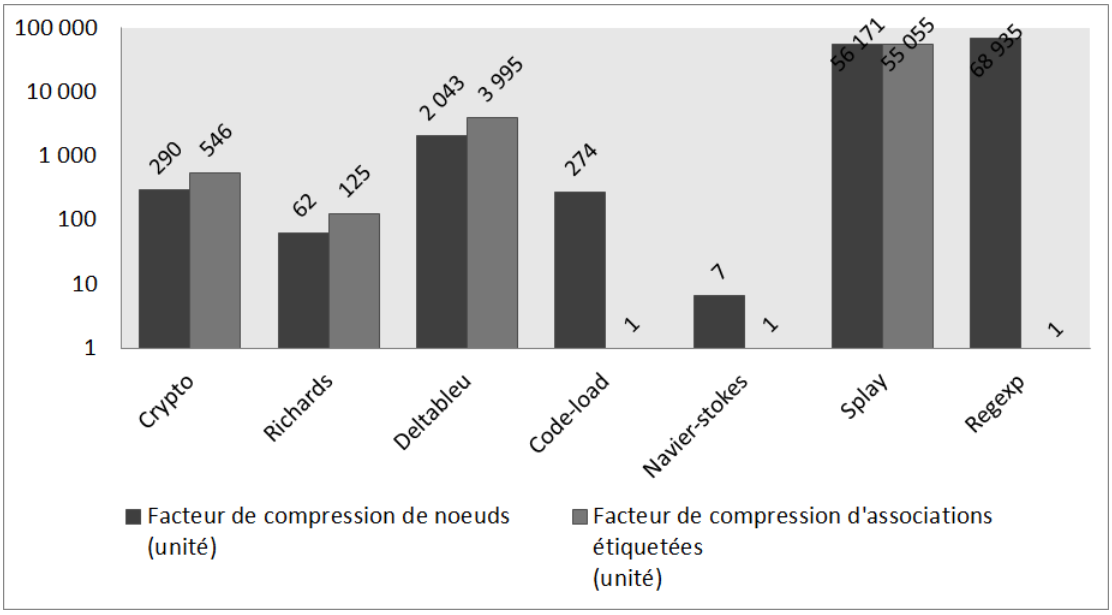


Figure 4.6 – Facteur de compression en comparant le GOC avec le graphe d'appartenances abstrait.

4.4 Étude de cas

Cette section présente une étude plus détaillée de quelques résultats produits par l'outil JSTI. Nous considérons les programmes *Splay*, *Crypto* et *Deltablue* qui illustrent des cas intéressants dans notre analyse. De plus, ces programmes ont la particularité de générer des graphes différents selon le mode d'analyse. Pour chacun de ces programmes, le graphe généré par JSTI est expliqué en fonction du code du programme.

4.4.1 Représentation visuelle des graphes d'objets abstraits

Les GOAs générés par JSTI utilisent le format Graphviz et peuvent être visualisés par une multiple d'applications qui supportent ce format très répandu. Pour faciliter la compréhension, différents attributs visuels sont utilisés pour représenter l'information contenue dans les graphes.

Un nœud est affiché comme un cercle ou rectangle. Les ovales représentent les objets internes de JS (par exemple, (*Array*, *String*, *RegExp*, etc.). Les autres nœuds sont représentés par des rectangles. Le style de la bordure d'un nœud indique l'importance d'un nœud (gras : nœud a une forte importance, noir : nœud a une moyenne importance, gris clair : nœud a une faible importance). Deux types d'arcs existent : les arcs continus non étiquetés, et les arcs étiquetés sous forme de tirets. Un arc continu non étiqueté représente le lien de prototype et un arc étiqueté représente un lien d'association.

Par exemple, le graphe de la figure 4.7 contient trois noeuds. Le premier est un ovale car il est du type interne `Date` de JS. Sa couleur gris clair indique sa faible importance. Deux noeuds rectangulaires correspondent à des types de l'utilisateur (`Animal` et `Chat`). L'un des deux est noir, qui indique sa moyenne importance, et l'autre gras ce qui montre sa forte importance. Le graphe contient aussi deux arcs : un continu sans étiquete qui représente la relation du prototype (`Animal` est le prototype du `Chat`). L'autre arc, étiqueté par `Born_Date`, représente le fait que la propriété `Born_Date` de la classe `Chat` est du type `Date`.

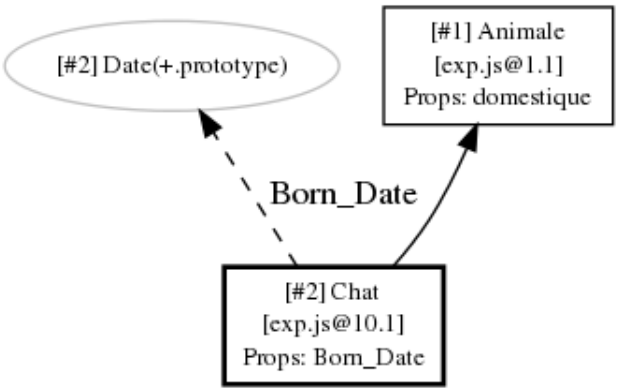


Figure 4.7 – Illustration du format d'un GOA

4.4.2 *Splay*

La figure 4.8 représente le graphe généré en mode en-ligne pour le programme *Splay*. Ce programme crée plus d'un million d'objets, tandis que son graphe de types ne contient que 21 nœuds, avec un facteur de compression de nœuds est égal *53,496* et un facteur de compression d'associations qui dépasse 100,000. Le nombre d'associations est réduit de 1,156,156 à 11 associations. La partie encadrée est détaillée dans la figure 4.9.

La figure 4.9 représente un fragment du graphe de types pour *Splay*. Ce fragment contient trois nœuds numérotés : un objet `Node` (1), son prototype (2) ainsi qu'un objet littéral auquel l'objet `Node` pointe par sa propriété `value`.

fr

Le constructeur de l'objet `Node` apparaît ci-dessous :

```
SplayTree.Node = function(key, value) {
  this.key = key;
  this.value = value;
};
```

On remarque que le constructeur ne définit que deux propriétés : `key` et `value`, tandis que le nœud du GOA en contient deux autres nommées `left` et `right`. Ces deux dernières propriétés ont été ajoutées par l'appel d'une autre fonction une fois l'objet `Node` créé :

```
SplayTree.prototype.splay_ = function(key) {
```

```
  ...
  dummy = left = right = new SplayTree.Node(null, null);
  ...
  left.right = current.left;
  right.left = current.right;
  ...
};

SplayTree.prototype.insert = function(key, value) {
  ...
  var node = new SplayTree.Node(key, value);
  if (...) {
    node.left = this.root_;
    node.right = this.root_.right;
    ...
  } else {
    node.right = this.root_;
    node.left = this.root_.left;
    ...
  }
  ...
};
```

Cet exemple illustre bien la difficulté associée à comprendre la structure des objets dans un programme JS lorsque celle-ci évolue au cours de l'exécution.

Le nœud 2 est le prototype du premier nœud. Sa structure a été définie par ce code :

```
SplayTree.Node.prototype.left = null;
SplayTree.Node.prototype.right = null;
SplayTree.Node.prototype.traverse_ = function(f) {
  ...
};
```

On remarque que l'information associée à ce nœud dans le GOA correpond parfaitement à la structure du code.

Finalement, le nœud 3 représente un objet litéral créé par le code suivante :

```
function GeneratePayloadTree(depth, tag) {
  if (depth == 0) {
    ...
  } else {
    return {
      left:  GeneratePayloadTree(depth - 1, tag),
      right: GeneratePayloadTree(depth - 1, tag)
    };
  }
}
```

Une des valeurs de retour possibles de ce code est un objet littéral possédant deux propriétés `left` et `right`.De plus, chacune de ces propriétés stocke une référence vers un objet de même type retourné par un appel récursif à `GeneratePayloadTree`. Ceci cause l'ajout d'un lien d'association vers le même nœud dans le graphe abstrait généré par JSTI, tel qu'observé à la figure 4.9.

L'association entre le nœud 1 et le nœud 3 est causée par le code suivant :

```
function InsertNewNode() {
  ...
  var payload = GeneratePayloadTree(kSplayTreePayloadDepth,
                    String(key));
  splayTree.insert(key, payload);
  ...
}
```

La fonction `InsertNewNode` obtient un objet par un appel à la fonction `GeneratePayloadTree` mentionnée précédemment. Cet objet est représenté dans notre abstraction par le nœud 3. Cet objet est alors passé à la méthode `insert`, qui crée un nouveau nœud et définit sa propriété `value` pour pointer vers l'objet passé en paramètre. Ceci explique l'association entre les nœuds 1 et 3 étiquetée par `value`. Le cas de *Splay* démontre donc que le graphe d'objets abstrait généré par JSTI correspond bien au programme analysé. L'abstraction est concise et précise, et capture adéquatement les concepts exprimés dans le fragment de code exploré.

4.4.3 *Crypto*

On présente deux fragments de graphes pour le programme *crypto* : le premier extrait du graphe de types avant la simplification (figure 4.10) et le deuxième extraite du graphe de types après la simplification du GOA (figure 4.11).

On remarque que la première figure contient *6* nœuds, tandis que la deuxième ne contient que *5*. Les deux nœuds `11` et `12` de la première figure ont été groupés dans le nœud `12` de la deuxième figure, où les propriétés et les méthodes de ce nœud sont l'union de ceux des deux nœuds de la première figure.

Cet exemple se démarque aussi par la grande quantité de méthodes pour le nœud *BigInteger*. Une inspection du code révèle que ces méthodes sont bien ajoutées par le programme au prototype du constructeur pour `BigInteger`. On remarque aussi que l'association entre les nœuds 23 et 12 est étiquetée par 7 propriétés distinctes. Ces associations sont le résultat de plus de *50* lignes de code, elles-mêmes incluses dans un dans une fonction dont la taille excède *150* lignes. Un extrait de ce code est reproduit ici :

```
// (public) Constructor
function BigInteger(a,b,c) {
  this.array = new Array();
  ...
}

// 23   "empty" RSA key constructor
function RSAKey() {
  this.n = null;
  this.e = 0;
  this.d = null;
  this.p = null;
  this.q = null;
  this.dmp1 = null;
  this.dmq1 = null;
  this.coeff = null;
}
```

```
RSAKey.prototype.generate = RSAGenerate;

// Generate a new random private key B bits long, using public expt E
function RSAGenerate(B,E) {
  ...
  this.e = parseInt(E,16);
  var ee = new BigInteger(E,16);
  ...
  this.p = new BigInteger(B-qs,1,rng);
  ...
  this.q = new BigInteger(qs,1,rng);
  ...
  var t = this.p;
  this.p = this.q;
  this.q = t;
  var p1 = this.p.subtract(BigInteger.ONE);
  var q1 = this.q.subtract(BigInteger.ONE);
  ...
  var phi = p1.multiply(q1);
  ...
  this.n = this.p.multiply(this.q);
  this.d = ee.modInverse(phi);
  this.dmp1 = this.d.mod(p1);
  this.dmq1 = this.d.mod(q1);
  this.coeff = this.q.modInverse(this.p);
  ...
}
```

Les objets du nœud `12` ont été créés par le constructeur `BigInteger`, tandis que les objets du nœud `23` ont été créés par le constructeur `RSAKey`. Le nœud `23` réfère au nœud `12` par ses propriétés. La fonction `RSAGenerate` effectue une multitude d'opérations sur ces objets, mais le modèle abstrait généré par JSTI en représente succinctement les effets encore une fois.

4.4.4 *Deltablue*

JSTI génère, dans certains cas, des résultats qui différent de l'intuition. C'est le cas pour le programme *Deltablue*. La figure 4.12 montre un extrait du graphe généré qui illustre ce cas. En JS, chaque objet ne possède qu'un seul objet prototype. De plus, tel que discuté précédemment, aucun objet ne change son prototype durant l'exécution de programme dans les programmes étudiés. Malgré tout, la figure 4.12 montre que les nœuds `1` et `2` sont tous les deux des prototypes du nœud `3`.

Cette situation s'explique comme suit. Nous remarquons que les nœuds `1` et `2` sont similaires à l'exception de la propriété `input` qui existe uniquement dans le nœud `2`. Les deux modèles de types des nœuds `1` et `3` ont été créés en premier. Le nœud `1` est le prototype de nœud `3`. Ensuite, un objet de nœud `1` change son modèle de type par l'ajout dynamique de la propriété `input`. Ce dernier cause la création de nœud `2` pour l'objet modifié. Ceci explique pourquoi le nœud `2` devient un autre prototype de nœud `3`. Il est important de préciser que cette situation ne peut survenir que dans le mode en-ligne.

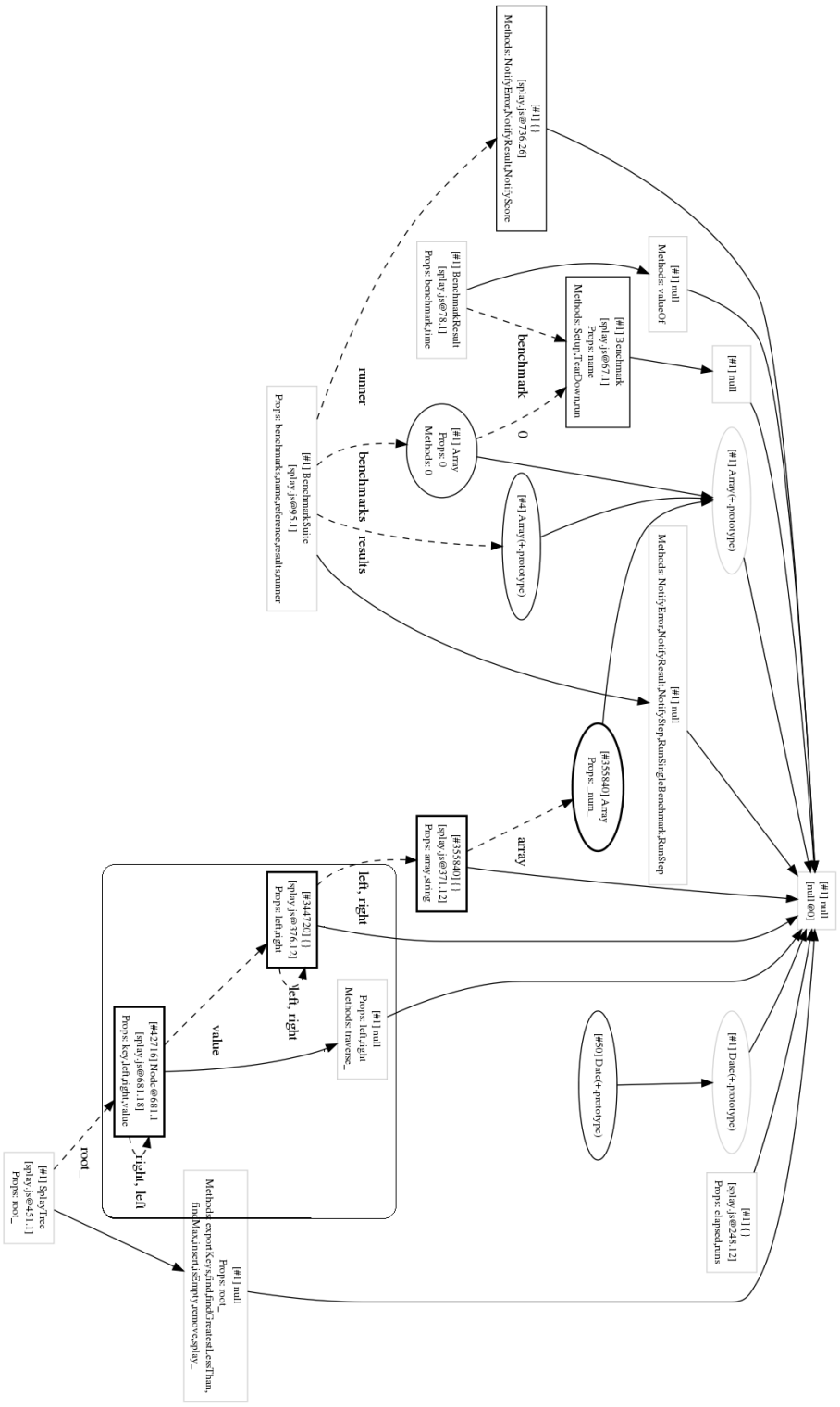


Figure 4.8 – Graphe de types pour *Splay* généré en mode en-ligne

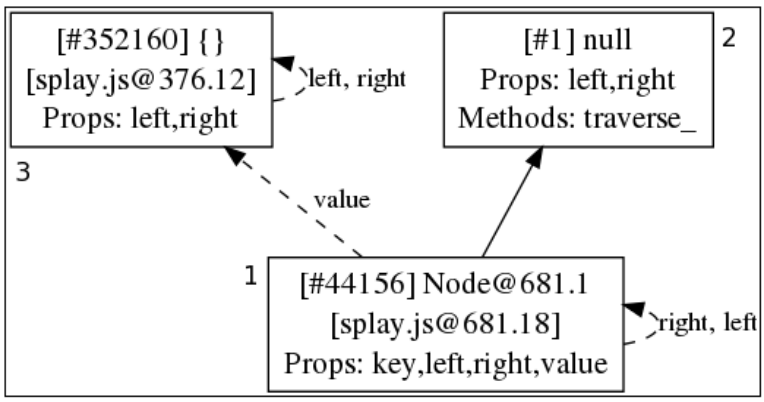


Figure 4.9 – Fragment de graphe de types pour *Splay*

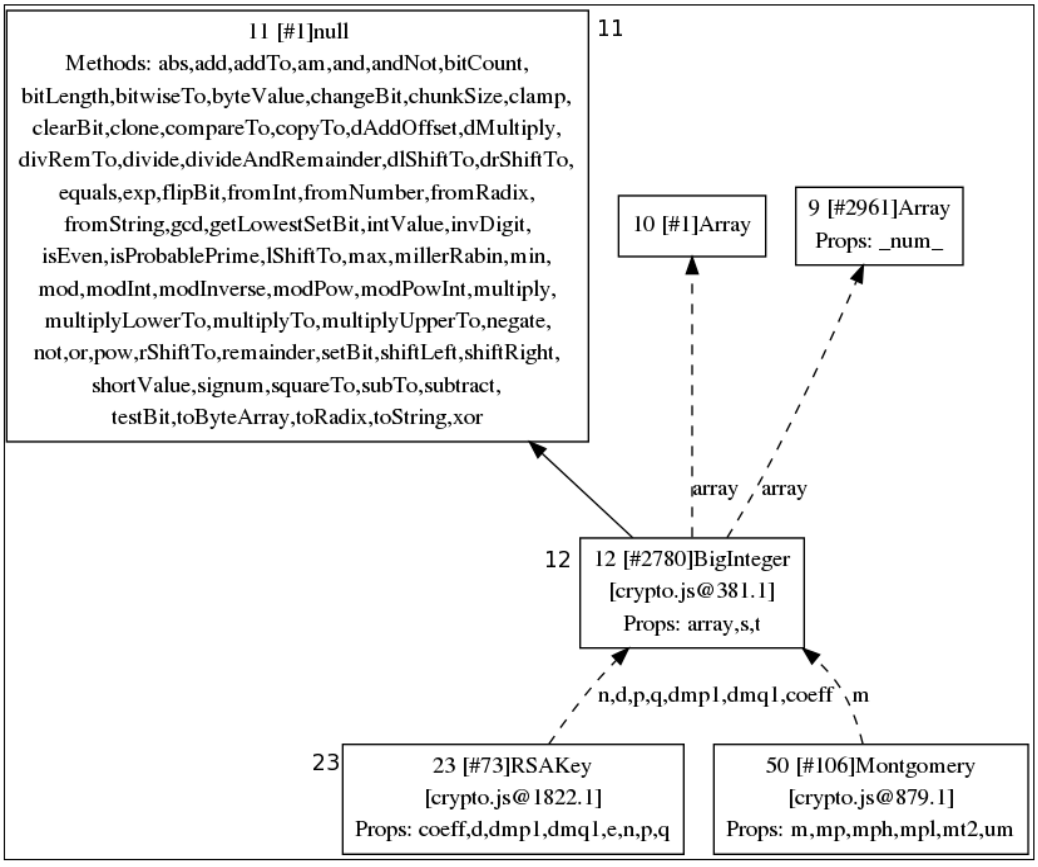


Figure 4.10 – Extrait de graphe de types pour *Crypto*

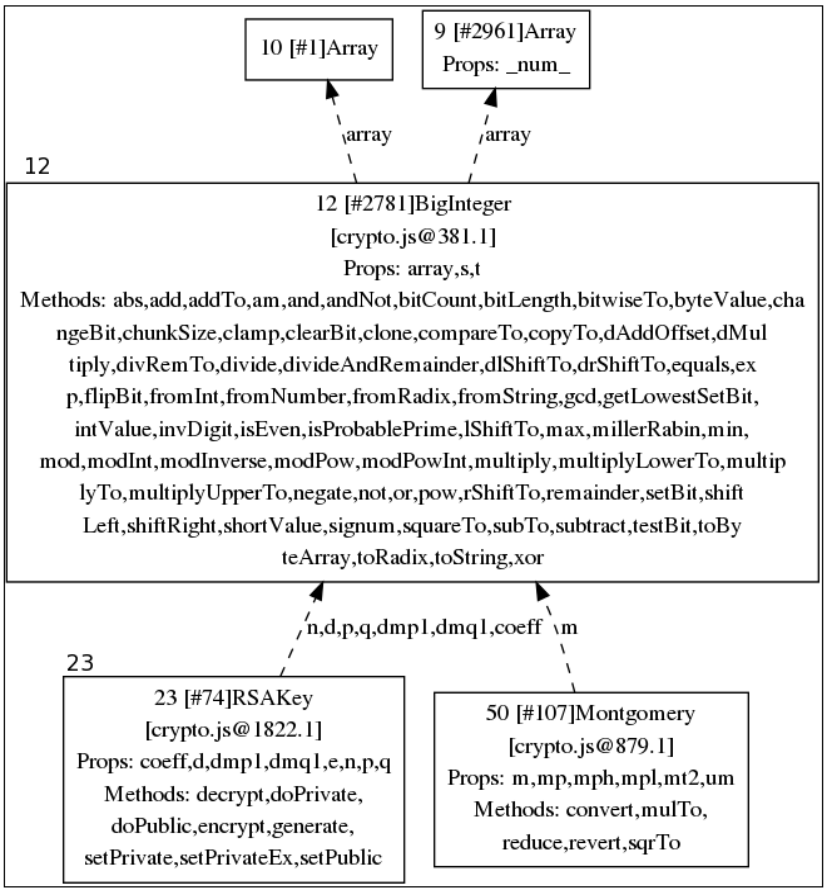


Figure 4.11 – Extrait de graphe de types abstrait pour *Crypto*

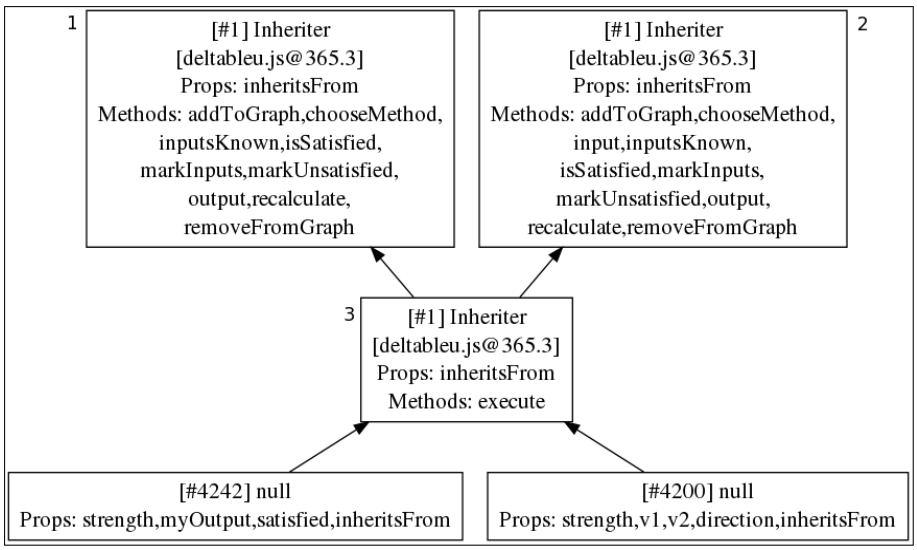


Figure 4.12 – Fragment de graphe de types pour *Deltablue* généré en mode en-ligne

CHAPITRE 5

CONCLUSION

La complexité du code des programmes actuels constitue un grand défi pour les développeurs d'applications JavaScript. L'analyse du comportement des objets et des liens entre eux est une tâche importante puisque la nature du langage les rend difficile à comprendre à partir du code source. Dans ce mémoire, nous avons proposé une approche dynamique qui se base sur la rétro-ingénerie pour analyser les programmes JS. Cette approche comporte trois volets : profiler les objets et les liens d'interaction, faire des abstractions pour générer un graphe d'objets abstrait (GOA), et finalement représenter ce GOA sous une forme visuelle inspirée du diagramme de classes d'UML.

Nous avons développé l'outil *JSTI* qui implémente les tâches de l'approche par différentes méthodes et supporte deux stratégies d'abstraction multiples. L'outil utilise l'instrumentation source à source pour effectuer le profilage. Pour assurer la complétude de l'information rapportée aux objets, deux types de données sont profilées : les informations reliées à la structure interne des objets ainsi que les dépendances entre eux. Deux types de dépendances sont gérées par notre approche : l'héritage et les associations entre objets.

Nous avons évalué la performance de notre approche ainsi que le taux de compression des abstractions générées à l'aide des dix programmes de références de V8. Les résultats obtenus montrent que notre approche est suffisamment efficace pour être utilisable en pratique avec un ralentissement moyen de 14x en comparaison avec le code original. De plus, les graphes abstraits générés sont souvent très compacts : pour 6 des 10 programmes étudiés, le graphe généré contenait moins de 30 nœuds au total.

5.1 Perspectives

Bien que les résultats préliminaires de notre approche sont prometteurs, il existe plusieurs opportunités d'amélioration. Certaines limites de l'approche ont été identifiées

dans le chapitre 3. À court terme, ces limites pourraient être corrigées sans grande difficulté. À moyen terme, d'autres possibilités d'extension existent.

5.1.1 Amélioration de la performance

Pour rendre JSTI plus utile en pratique, sa performance pourrait être améliorée. L'utilisation d'une instrumentation qui permet d'analyser une portion sélectionnée du code et non pas forcément tout le code est une avenue possible. Dans les applications web, il n'est pas nécessaire d'analyser plusieurs bibliothèques communément utilisées telles que jQuery. De tels filtres pourraient être utiles pour permettre à JSTI de gérer des programmes plus complexes et qui créent un grand nombre d'objets.

5.1.2 Exposition de l'information

Notre outil génère un graphe qui présente les données profilées. La façon d'exposer ces données à un développeur pourrait être plus intuitive et efficace. Cette amélioration se base sur l'ajout d'une relation entre le graphe et le code source. On pourrait, par exemple, afficher la partie du code qui a causée la construction d'un nœud sélectionné, ou même de déterminer les objets qui ont été construits par une partie précise du code. Une intégration avec un environnement de développement tel qu'Eclipse serait particulièrement utile pour atteindre cet objectif.

Un deuxième ajout utile à la présentation du graphe est de le rendre interactif. Plutôt que de présenter toutes les informations dans le graphe dès le départ, la structure du graphe pourrait être révélée de façon interactive selon les explorations d'un développeur.

5.1.3 L'application de l'approche hors JS

Notre approche pourrait aussi être utilisée pour d'autres langages qui ont des caractéristiques communes avec JS tels que Self et Python. Bien que Python possède la notion de classe, ce langage permet comme JS de modifier la structure interne d'un objet une fois créé, et souffre donc des mêmes problèmes que JS quant à la facilité de comprendre le code source.

Nous espérons de faire évoluer *JSTI* vers un outil assurant la génération de la bonne hiérarchie avec de meilleures performances mêmes dans des cas complexes, et assurant la complétude, la précision des informations fournies, la simplicité et la fiabilité de l'exposition des données.

BIBLIOGRAPHIE

[1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su et Samuel Z. Guyer. Heapviz : interactive heap visualization for program understanding and debugging. Dans *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 53–62, 2010. ISBN 978-1-4503-0028-5.

[2] Ole Agesen, Jens Palsberg et Michael I. Schwartzbach. Type Inference of Self. Dans *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 247–267, 1993. ISBN 3-540-57120-5.

[3] Christopher Anderson, Paola Giannini et Sophia Drossopoulou. Towards type inference for JavaScript. Dans *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 428–452, 2005. ISBN 3-540-27992-X, 978-3-540-27992-1.

[4] Alan H. Borning et Daniel H. H. Ingalls. A type declaration and inference system for smalltalk. Dans *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 133–141, 1982. ISBN 0-89791-065-6.

[5] Luca Cardelli. A semantics of multiple inheritance. Dans *Proc. of the international symposium on Semantics of data types*, pages 51–67, 1984. ISBN 3-540-13346-1.

[6] Robert Cartwright et Mike Fagan. Soft Typing. Dans David S. Wise, éditeur, *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 278–292. ACM. ISBN 0-89791-428-7.

[7] Stephen Chong et Radu Rugina. Static analysis of accessed regions in recursive data structures. Dans *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 463–482, 2003. ISBN 3-540-40325-6.

[8] Ravi Chugh et Ranjit Jhala. Dependent Types for JavaScript. *CoRR*.

[9] CoffeScript. Language compiles into JavaScript. (dernière visite : 8 août 2012). URL `http://coffeescript.org/`.

[10] P. Cousot et R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Dans *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[11] Werner Dietl et Peter Müller. Runtime universe type inference. Dans *T. Wrigstad*, IWACO '07, pages 3–15, Berlin, Germany, July 2007. Proceedings of the International Workshop on Aliasing, Confinement and Ownership in object-oriented programming.

[12] Cormac Flanagan et Stephen N. Freund. Dynamic architecture extraction. Dans *Proceedings of the First combined international conference on Formal Approaches to Software Testing and Runtime Verification*, FATES'06/RV'06, pages 209–224, 2006. ISBN 3-540-49699-8, 978-3-540-49699-1.

[13] Rakesh Ghiya et Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph ? A Shape Analysis for Heap-Directed Pointers in C. Dans Hans-Juergen Boehm et Guy L. Steele Jr., éditeurs, *POPL*, pages 1–15. ACM Press. ISBN 0-89791-769-3.

[14] Justin O. Graver et Ralph E. Johnson. A Type System for Smalltalk. Dans Frances E. Allen, éditeur, *POPL*, pages 136–150. ACM Press. ISBN 0-89791-343-4.

[15] Justin O. Graver et Ralph E. Johnson. A type system for Smalltalk. Dans *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 136–150, 1990. ISBN 0-89791-343-4.

[16] Justin Owen Graver. *Type checking and type inference for object-oriented programming languages*. Thèse de doctorat, Champaign, IL, USA, 1989. AAI9010868.

[17] Trent Hill, James Noble et John Potter. Scalable Visualisations with Ownership Trees. Dans *TOOLS (37)*, pages 202–213. IEEE Computer Society, .

[18] Trent Hill, James Noble et John Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput.*, (3):319–339, .

[19] D. Ingalls. *The Execution Time Profile as a Programming Tool ; Design and Optimization of Compilers*. Prentice-Hall, Englewood Cliffs, 1971.

[20] ECMA International. ECMAScript language specification. Stardard ECMA-262, 3rd Edition, 1999.

[21] Simon Holm Jensen, Anders Møller et Peter Thiemann. Type Analysis for JavaScript. Dans *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, 2009. ISBN 978-3-642-03236-3.

[22] Ralph E. Johnson. Type-checking Smalltalk. *SIGPLAN Not.*, 21(11):315–321, juin 1986. ISSN 0362-1340.

[23] Marc A. Kaplan et Jeffrey D. Ullman. A general scheme for the automatic inference of variable types. Dans *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 60–75, 1978.

[24] Mark Marron, Cesar Sanchez, Zhendong Su et Manuel Fähndrich. Abstracting Runtime Heaps for Program Understanding. *CoRR*.

[25] Robin Milner. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, (3):348–375.

[26] Salman Mirghasemi, John J. Barton et Claude Petitpierre. Naming anonymous JavaScript functions. Dans Cristina Videira Lopes et Kathleen Fisher, éditeurs, *OOPSLA Companion*, pages 277–288. ACM. ISBN 978-1-4503-0942-4.

[27] Prateek Mishra et Uday S. Reddy. Declaration-free type checking. Dans *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '85, pages 7–21, 1985. ISBN 0-89791-147-4.

[28] Nick Mitchell, Edith Schonberg et Gary Sevitsky. Making Sense of Large Heaps. Dans *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 77–97, 2009. ISBN 978-3-642-03012-3.

[29] Node.js. JavaScript platform. (dernière visite : 8 août 2012). URL `http://nodejs.org/`.

[30] Jens Palsberg et Michael I. Schwartzbach. Object-oriented type inference. Dans *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 146–161, 1991. ISBN 0-201-55417-8.

[31] Jens Palsberg et Tian Zhao. Type inference for record concatenation and subtyping. *Inf. Comput.*, 189(1):54–86, février 2004. ISSN 0890-5401.

[32] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides et Jeaha Yang. Visualizing the Execution of Java Programs. Dans *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2002. ISBN 3-540-43323-6.

[33] Wim De Pauw et Gary Sevitsky. Visualizing Reference Patterns for Solving Memory Leaks in Java. Dans *2007 Future of Software Engineering*, FOSE '07, pages 104–119, 2007. ISBN 0-7695-2829-5.

[34] Sokhom Pheng et Clark Verbrugge. Dynamic Data Structure Analysis for Java Programs. Dans *ICPC*, pages 191–201. IEEE Computer Society. ISBN 0-7695-2601-2.

[35] Easwaran Raman et David I. August. Recursive data structure profiling. Dans *Proceedings of the 2005 workshop on Memory system performance*, MSP '05, pages 5–14, 2005. ISBN 1-59593-147-3.

[36] Derek Rayside et Lucy Mendel. Object ownership profiling : a technique for finding and fixing memory leaks. Dans *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 194–203, 2007. ISBN 978-1-59593-882-4.

[37] Derek Rayside, Lucy Mendel et Daniel Jackson. A dynamic analysis for revealing object ownership and sharing. Dans *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, pages 57–64, 2006. ISBN 1-59593-400-6.

[38] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky et Saher Esmeir. BrowserShield : Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3), septembre 2007. ISSN 1559-1131.

[39] Mooly Sagiv, Thomas Reps et Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. Dans *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 16–31, 1996. ISBN 0-89791-769-3.

[40] Mooly Sagiv, Thomas Reps et Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. Dans *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 105–118, 1999. ISBN 1-58113-095-3.

[41] Norihisa Suzuki. Inferring types in Smalltalk. Dans *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 187–199, 1981. ISBN 0-89791-029-X.

[42] Peter Thiemann. Towards a type system for analyzing JavaScript programs. Dans *Proceedings of the 14th European conference on Programming Languages and Systems*, ESOP'05, pages 408–422, 2005. ISBN 3-540-25435-8, 978-3-540-25435-5.

[43] Andrew K. Wright et Robert Cartwright. A Practical Soft Type System for Scheme. *ACM Trans. Program. Lang. Syst.*, (1):87–152.