Génération automatique de tests de la spécification JAVASCRIPT

Benjamin Farinier

Table des matières

I Le projet JSCERT **1**

1 ECMASCRIPT5 et la sémantique JAVASCRIPT **2**
1.1 Expression, instruction et programme . 2
1.2 Objets et prototypage 3
1.3 État, environnement et contexte 4

2 JSCERT et JSREF **5**
2.1 JSCERT 5
2.2 JSREF 6
2.3 Fiabilité de la formalisation 6

II Outillage **6**

3 BISECT **7**

4 Top-level **8**
4.1 Structure du top-level 8
4.2 Quelques fonctionnalités 8

5 Générateur de tests **9**
5.1 Fonctionnement du générateur 9
5.2 Critiques 10

Introduction

JAVASCRIPT est un des langages de script les plus utilisés pour étendre les fonctionnalités de pages web. On le rencontre par exemple dans de nombreux outils Google (tels Google Maps ou Gmail). Bien qu'il possède une spécification standardisée, ECMASCRIPT5, sa sémantique n'est pas donnée de manière formelle. Les implémentations se basent donc sur des jeux de tests pour montrer leur adéquation à la spécification.

L'équipe du projet JSCERT travaille sur la formalisation en COQ de la sémantique de JAVASCRIPT. De cette formalisation, ils extraient un interpréteur de JAVASCRIPT qu'ils peuvent donc éprouver avec les jeux de tests. Ils ont récemment découvert qu'aucun navigateur n'implémente correctement certaines fonctionnalités, comme les instructions `try` **Block** `finally` **Finally**, très probablement parce que ces fonctionnalités ne sont pas testées. C'est de ce constat que vient la motivation de ce stage : pouvoir générer des tests sur cette spécification.

Dans une première partie je commence par expliquer le fonctionnement du langage JAVASCRIPT avant de présenter les composantes du projet JSCERT avec lesquelles j'ai interagi durant mon stage. Je présente ensuite dans une seconde partie le travail que j'ai réalisé au cours de mon stage au sein de l'équipe Celtique de l'Inria de Rennes, sous la direction de mon maître de stage Alan Schmitt.

Première partie

Le projet JSCERT

Le langage JAVASCRIPT a été initialement développé afin d'enrichir les pages web en permettant l'exécution de scripts par les navigateurs internet. Il est aujourd'hui utilisé de façon prédominante sur le web non seulement pour ajouter de l'interactivité aux

sites web, mais aussi en tant que plate-forme pour le développement d'applications écrites dans d'autres langages.

En plus de son adoption massive, JAVASCRIPT présente deux caractéristiques importantes. Premièrement, ayant été développé dans le but de faciliter son intégration dans les navigateurs et le contenu web, il vise plutôt à fournir des fonctionnalités puissantes qu'à donner des garanties de robustesse et de sécurité. Une seconde caractéristique de JAVASCRIPT est sa standardisation, fournissant de nombreuses informations sur la manière dont sont supposées interagir ces fonctionnalités.

Le but du projet JSCERT[1] est de produire une sémantique formelle et précise de JAVASCRIPT afin de pouvoir construire des outils certifiés pour l'analyse et la compilation. Ce projet est une collaboration entre les équipes de l'Inria et de l'Imperial College.

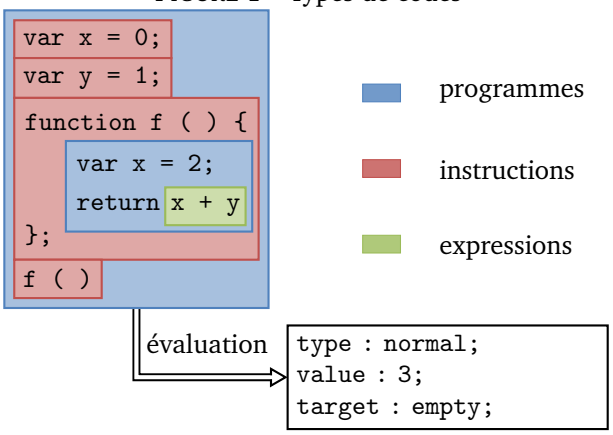
1 ECMASCRIPT5 et la sémantique JAVASCRIPT

Le standard ECMASCRIPT5[2] définit le langage standardisé JAVASCRIPT. Les navigateurs existant implémentent majoritairement ce standard, mais ajoutent parfois certaines extensions ou font différents choix dans l'implémentation de certains cas spécifiés comme étant *implementation dependent*. En plus du langage, ce standard décrit un ensemble de librairies proposant plusieurs fonctionnalités utiles. Les navigateurs les plus récent adhèrent généralement au standard ECMASCRIPT5, du moins en ce qui concerne les caractéristiques du langage. À quelques exceptions près, la spécification ECMASCRIPT5 est très précise et non ambiguë.

1.1 Expression, instruction et programme

La grammaire de JAVASCRIPT est définie inductivement et est divisée en trois catégories : les expressions, les instructions et les programmes. Elle comprend 13 types d'expressions et 17 types d'instructions. La définition des expressions dépend de la grammaires des littéraux (`null`, `bool`, `number` et `string`), de 11 opérateurs unaires et de 24 opérateurs binaires.

FIGURE 1 – Types de codes



Programme et code exécuté

Un programme JAVASCRIPT consiste en une liste d'instructions ou de fonctions, accompagnée d'un indicateur influençant sur sa sémantique d'évaluation (mode *strict* ou non). Le corps d'une fonction est définie comme étant lui même un programme JAVASCRIPT, comme dans l'exemple de la figure 1. Similairement, les arguments d'un appel à `eval` sont eux aussi considérés comme des programmes. Ces trois types de programmes rejoignent les trois types de code exécutables prévus par la spécification ECMASCRIPT5 :

- Le *code global* est le texte source traité comme un programme. Il n'inclut pas les parties du source décrivant le corps des fonctions.
- Le *code évalué* est le texte source fourni à la fonction `eval`. Plus précisément, si le paramètre de la fonction `eval` est une chaîne de caractères, cette dernière est traitée comme un programme dont elle est le code global.
- Le *code fonctionnel* est le texte source décrivant le corps d'une fonction. Il n'inclut pas les parties du source décrivant les fonctions imbriquées.

Triplet de complétion

Le résultat de l'évaluation d'une instruction ou d'un programme est un triplet dit de complétion. Un triplet de complétion est constitué d'un *type*, d'une *valeur* optionnelle (ou une référence si l'expression est un nom de variable) et d'une *étiquette* optionnelle. La valeur d'un triplet de complétion peut être utilisé. Ainsi, le code `y = eval ("do var x = 2 while false")` positionne la valeur de la variable `y` à 2.

Le type, `Normal`, `Return`, `Break`, `Continue` ou `Throw`, correspond à la directive de contrôle de flot obtenue lors de l'évaluation. Une évaluation renvoyant un type autre que `Normal` est qualifiée de *abrupte*. La valeur, si elle existe, est soit une valeur primitive (un littéral ou `undefined`), soit l'emplacement d'un objet alloué. Elle décrit le résultat d'un expression ou d'une instruction (type `Normal`), une valeur renvoyée par une instruction (type `Return`)) ou une exception levée (type `Throw`). Enfin l'étiquette est seulement utilisée pour les types `Break` et `Continue`, dans le but d'implémenter les instructions éponymes avec labels.

Une particularité de JAVASCRIPT est que les types `Break` et `Continue` peuvent aussi être associés à des valeurs de retour.

1.2 Objets et prototypage

ECMASCRIPT5 décrivant un langage orienté objet de haut niveau, il fonctionne avec des objets. Un objet est défini comme étant une *collection de propriétés* qui possède un *unique objet prototype*. Le prototype peut avoir pour valeur soit un objet soit `null`.

Voici un exemple simple :

```
var obj = {
  x : 1,
  f : function () {return this.x}
}
```

Dans cet exemple, l'objet `obj` est déclaré avec deux propriétés explicites, `x` ayant pour valeur `1` et `f` une fonction retournant la valeur de `x`, ainsi qu'une propriété implicite *@proto* qui est une référence vers le prototype de l'objet `obj`.

Chaîne de prototypes

Les prototypes sont des objets comme les autres et peuvent posséder leurs propres prototypes. Un prototype peut posséder une référence vers un autre prototype et ainsi de suite. Lorsque plusieurs prototypes sont chaînés par des références non-nulles on dit qu'il s'agit d'une *chaîne de prototypes*. Une chaîne de prototypes est une chaîne finie d'objets utilisée pour implémenter l'héritage et le partage de propriétés. On parle d'héritage prototypal.

Par exemple :

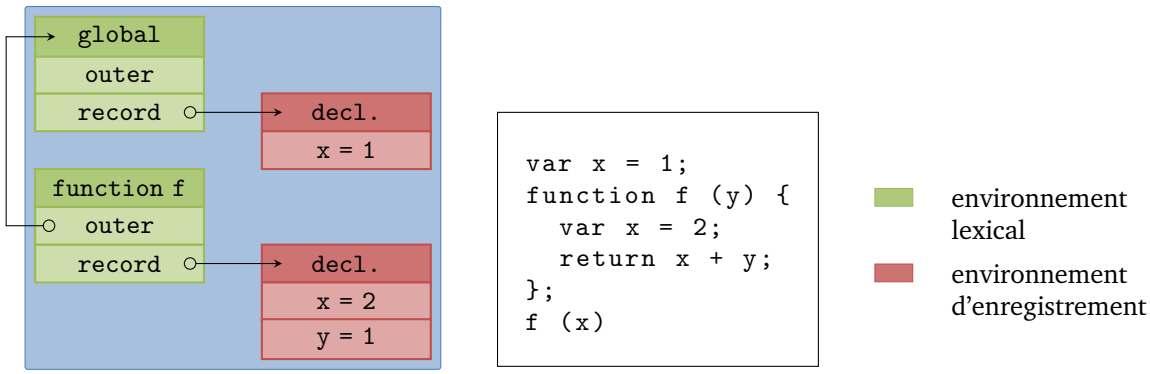
```
var base = {
  y : 2,
  f : function () {
    return this.x + this.y
  }
}

var obj1 = {
  __proto__ : base,
  x : 0
}

var obj2 = {
  __proto__ : base,
  x : 1
}
```

Dans cette exemple les objets `obj1` et `obj2` héritent tout deux de l'objet `base`. Cependant dans le premier la méthode `f` utilise `x` avec `0` comme valeur tandis que le second avec `1` comme valeur.

Résolution du chaînage

La règle de résolution est simple : si une propriété ou une méthode n'est pas trouvée dans l'objet lui-même, elle sera recherchée dans la chaîne de prototype. Si la propriété n'est pas trouvée dans le prototype, alors une recherche aura lieu dans le prototype du prototype et ainsi de suite. La chaîne de prototype est donc remontée, et c'est la première propriété ou méthode trouvée ayant le même nom qui sera utilisée. Une propriété trouvée est appelée propriété héritée. Si la propriété n'est pas trouvée après le parcours intégral de la chaîne de prototype, la valeur `undefined` sera retournée.

FIGURE 2 – État du contexte d'exécution lors de l'appel d'une fonction



Notons que lorsque l'on accède à une méthode héritée, la valeur de `this` est égale au contexte de l'objet original (la notion de contexte est expliquée plus loin), et non au contexte du prototype de l'objet dans lequel la méthode a été trouvée. Dans l'exemple ci-dessus `this.x` est récupéré depuis `obj1` et `obj2` mais pas `base`, alors que `this.y` est récupéré depuis `base` grâce au mécanisme de chaînage prototypal.

Objets spéciaux

Certains objets spéciaux ont une utilisation particulière. Par exemple l'objet global qui stocke les variables globales et les fonctions innées du langage. En tant qu'objet, son champ *@proto* est lié à l'objet prototype. Ce deuxième objet spécial est le prototype de tous les objets. Si un prototype n'est pas spécifié explicitement lors de la création d'un objet, alors la valeur par défaut de *@proto* sera assignée à `Object.prototype`. L'objet prototype est le dernier maillon de la chaîne des prototypes et possède lui aussi une propriété *@proto* qui a pour valeur `null`. Enfin, chaque fonction contient un prototype `Function.prototype`, un autre objet spécial équipé de méthodes spécifiques à la fonction.

1.3 État, environnement et contexte

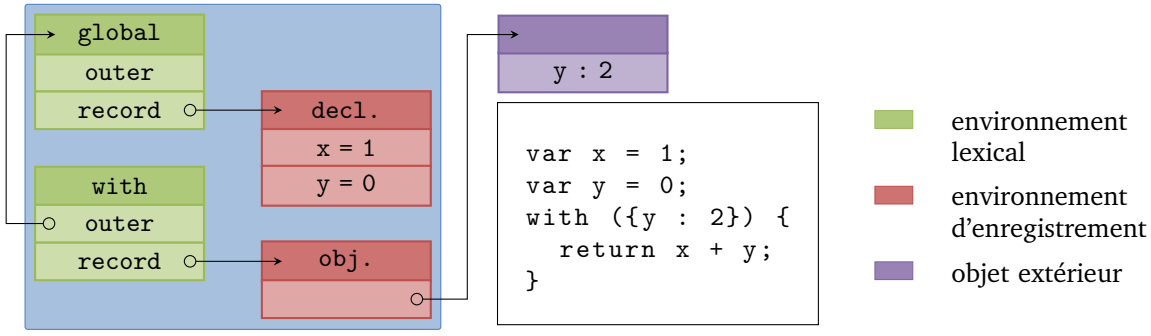
Un programme JAVASCRIPT est toujours exécuté dans un état et un contexte d'exécution donnés. L'état consiste en un ensemble d'objets alloués, plus un ensemble d'*environnements*. Le contexte d'exécution est décrit plus loin.

Environnements lexicaux et environnements d'enregistrement

Les environnements lexicaux servent à définir les associations entre identifieurs et variables ou fonctions en respect avec la structure imbriquée d'un code JAVASCRIPT. Un environnement lexical est composé d'un environnement d'enregistrement est d'une référence (potentiellement `null`) vers un environnement lexical extérieur. Les environnements lexicaux sont généralement associés à certaines structures spécifiques, tels que la déclaration de fonctions (figure 2), l'instruction `with` (figure 3) ou la clause `catch` de l'instruction `try`. Un nouvel environnement lexical est créé à chaque fois que ces instructions sont évaluées.

Les environnements d'enregistrement mémorisent les liaisons d'identifications créées dans son environnement lexical associé. Ces environnements peuvent prendre deux formes : un *environnement de déclaration* qui associe aux noms de variable leur valeur, ou un *environnement d'objet* qui contient l'emplacement d'un objet JAVASCRIPT et est utilisé dans l'implémentation de la sémantique de la portée associée au constructeur `with`. Par exemple dans la figure 2, l'environnement de déclaration de la fonction `f`

FIGURE 3 – État du contexte d'exécution lors de l'évaluation d'un `with`



contient les assignations pour les variables `x` et `y`.

La référence vers l'environnement extérieur est utilisée pour représenter l'imbriquement logique des environnements lexicaux. L'environnement extérieur englobe logiquement l'environnement intérieur lors de la résolution des portées. Lorsqu'un identifiant est évalué, on remonte cette chaîne de portées jusqu'au premier environnement contenant une assignation à l'identifiant cherché. Ainsi dans l'exemple de la figure 3, c'est la valeur de l'environnement interne à l'objet qui est utilisée pour l'assignation de sa valeur à la variable `y`.

Un environnement lexical peut servir d'environnement extérieur pour plusieurs environnements intérieurs, par exemple lorsqu'une déclaration de fonction contient dans son corps deux déclarations de fonctions.

Contexte d'exécution

Les contextes d'exécutions forment une pile dont le sommet est le contexte d'exécution courant. Un nouveau contexte d'exécution est créé à chaque fois que l'exécution passe d'un code exécutable à un autre. Le contexte nouvellement créé est alors placé au sommet de la pile et devient le contexte d'exécution courant.

Un contexte d'exécution contient toutes les données nécessaires à l'exécution du code associé. En particulier, il contient un environnement lexical, un environnement de variables et un lien vers la valeur `this`.

L'environnement lexical et l'environnement de variables sont des environnements lexicaux. Quand un contexte d'exécution est créé, ces deux environnements sont initialisés à l'identique. L'environnement lexical peut être amené à changer au cours de l'exécution tandis que l'environnement de variables n'est jamais modifié.

Dans la plupart des cas, seul le contexte d'exécution au sommet de la pile est manipulé par les programmes.

2 JSCERT et JSREF

2.1 JSCERT

JSCERT formalise la sémantique de JAVASCRIPT précédemment décrite à l'aide de jugement de la forme $\overline{t, S, C \Downarrow o}$, où t est une instruction (des jugements similaires existent pour les expressions et programmes), S un état, C un contexte et o une sortie. Pour une exécution qui termine, la sortie est une paire faite de l'état final et du triplet de complétion produit par l'évaluation. La sémantique de JSCERT est aussi capable de caractériser les exécutions divergentes.

La sémantique utilisée par JSCERT est une sémantique dite *pretty-big-step*[3] développée par Arthur Charguéraud. La différence principale avec une sémantique à grand pas réside dans l'utilisation de

formes intermédiaires dans la décomposition de l'évaluation. Ce style de sémantique permet de satisfaire plus efficacement à la norme ECMAScript5.

En effet, comme la plupart des spécifications de langages de programmation industriels, ECMAScript5 relie l'évaluation d'un terme directement à son résultat avec des phrases de la forme "*Let R be the result of evaluating t*", comme un jugement de sémantique à grand pas le ferait. D'un autre coté, utiliser la sémantique à grand pas traditionnelle entraîne de sérieux problèmes de duplication, qui apparaissent dès que l'on veut représenter des exceptions ou la divergence. En particulier, de nombreuses prémices ont besoin d'être copiées-collées dans plusieurs règles d'évaluation.

Le problème ici est que les pas de la sémantique à grand pas sont trop grands pour pouvoir détailler suffisamment la spécification, et son emploi avec ECMAScript5 entraînerait une explosion de la taille des règles. La sémantique *pretty-big-step* permet de suivre la structure de ECMAScript5 tout en évitant cette duplication.

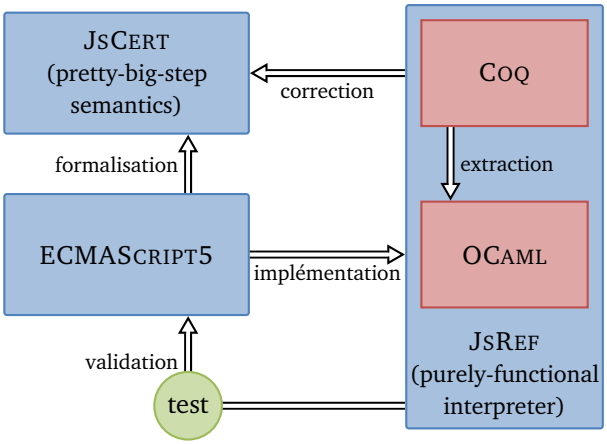
2.2 JsRef

JsRef est un interpréteur JavaScript écrit en Coq puis extrait vers OCaml. Son but premier est de suivre au plus près la spécification fournit par ECMAScript5. La performance n'est pas le but recherché : il doit seulement être suffisamment rapide pour pouvoir interpréter l'ensemble des programmes de la suite de tests JavaScript.

JsRef est écrit en Coq, un langage purement fonctionnel qui n'admet que des fonctions totales. ECMAScript5 est écrit en style impératif, admettant l'existence d'une mémoire globale mutable, et ses procédures peuvent abruptement retourner un résultat en cours d'exécution, ce qui aurait donné sens à l'emploi d'un langage impératif pour l'implémentation d'un interpréteur.

Cependant, même si utiliser un langage proche de JavaScript faciliterait l'implémentation de l'interpréteur, cela rendrait l'argument selon lequel cette mise en œuvre correspond à la sémantique formelle de JsRef beaucoup plus difficile. En effet, le fait que les fonctionnalités se ressemblent ne garantit pas que

FIGURE 4 – Interactions des différentes parties du projet JsCert



leur exécutions satisfassent les même contraintes. L'utilisation de Coq comme langage de programmation permet de rendre explicite l'état mémoire et le mécanisme de propagation des exceptions, tout en permettant d'exprimer plus facilement la preuve de son implémentation.

2.3 Fiabilité de la formalisation

Il est essentiel que la sémantique transcrive de façon fiable celle décrite par ECMAScript5. Comment se convaincre que c'est réellement le cas ? D'un coté JsCert suit au plus près la structure de ECMAScript5, il est donc possible de les comparer côte à côte. De l'autre coté, il est possible de lancer JsRef sur les suites de tests JavaScript contenant un grand nombre de programmes couvrant de nombreux aspects de la spécification. De plus, JsCert et JsRef sont reliés par une preuve Coq de correction. Cette correction assure que si l'interpréteur JsRef exécute un programme JavaScript et renvoie un résultat, alors le programme considéré renvoie ce résultat en suivant la sémantique de JsCert.

Deuxième partie

Outillage

Mon travail s'est divisé en deux parties : l'intégration à la compilation d'outils préexistants au projet JSCERT, puis la création de nouveaux afin d'aider au développement et de rendre plus fiable l'interpréteur JSREF.

Les outils préexistants intégrés sont au nombre de deux : OCAMLDEBUG et BISECT. OCAMLDEBUG est l'outil de debuggage présent dans la suite OCAML, tandis que BISECT est un outil de couverture. Ces deux outils permettent de faciliter la recherche d'erreurs dans l'implémentation de l'interpréteur JSREF. Pour tout deux, mon travail a consisté en leur insertion par de lourdes modifications du `Makefile`.

Les outils créés sont aussi au nombre de deux : un top-level pour l'interpréteur JSREF et un générateur de tests pour la spécification JAVASCRIPT. Le premier permet une utilisation interactive de l'interpréteur, tandis que le second permet de générer des tests couvrant certains points non vérifiés par la suite de tests officielle. J'ai choisi OCAML comme langage pour l'implémentation de ces deux outils.

Bien que très utile, je ne parlerai pas de OCAMLDEBUG par le suite, car il est inséré tel quel au projet JSCERT.

3 BISECT

BISECT[4] est un outil de couverture de code pour le langage OCAML développé par Xavier Clerc. Son nom est tiré de l'acronyme *Bisect is an Insanely Short-sized and Elementary Coverage Tool.*

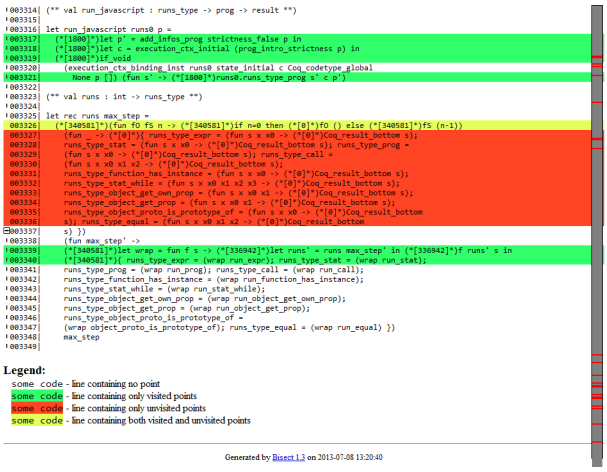
La couverture de code est une méthode pour le test de programmes pouvant avoir plusieurs utilisations. Le but premier de la couverture de code est de mesurer la portion du code source d'une application qui est mise à l'épreuve par un jeu de tests. Elle peut aussi être utilisée pour vérifier que l'exécution du programme sur une entrée donnée passe bien par les branches voulues, ou encore pour évaluer la qualité d'un jeu de tests.

Pour ce faire, l'outil de couverture définit des *points* dans le code source et mémorise si pendant l'exécution des tests le programme passe par cette branche du code. Les emplacements de ces points sont choisis afin d'assurer que toutes les alternatives ont été testées (par example, les branches d'un `if` ou d'un `match` sont contrôlées). En pratique :

- Premièrement, l'application est *instrumentée* : cela signifie que la forme compilée du programme est enrichie de sorte à pouvoir compter le nombre de fois où il passe par un point lors de l'exécution.
- Ensuite, le programme est lancé sur un jeu de tests et produit des données d'exécution sur la couverture.
- Enfin, un rapport est généré à partir des données produites lors de l'étape précédente, indiquant quels points ont été contrôlés durant les tests.

La couverture de code est une mesure utile mais, étant basée sur des tests, elle ne peut assurer la correction d'un programme. Elle donne seulement des indications pour la recherche de code non testé, voire mort.

FIGURE 5 – Rapport de couverture BISECT



4 Top-level

Un top-level est un programme permettant l'utilisation interactive d'un interpréteur pour un certain langage de programmation. Il permet l'appel d'instructions pas à pas, tout en conservant l'état d'exécution dans lequel il se trouve.

4.1 Structure du top-level

Le code du top-level se décompose en deux parties. L'une s'occupe de l'interaction avec l'utilisateur et l'autre de l'interaction avec le programme.

On a vu qu'un programme JAVASCRIPT s'exécute dans un état et un contexte d'exécution donné. Lors de l'exécution d'un programme JAVASCRIPT, ces deux éléments sont amenés à évoluer. Dans un programme classique, l'évaluation se faisant d'un bloc, il n'est pas nécessaire de conserver l'état et le contexte d'exécution une fois le programme terminé. Dans un top-level au contraire, on veut que les instructions précédemment exécutées persistent.

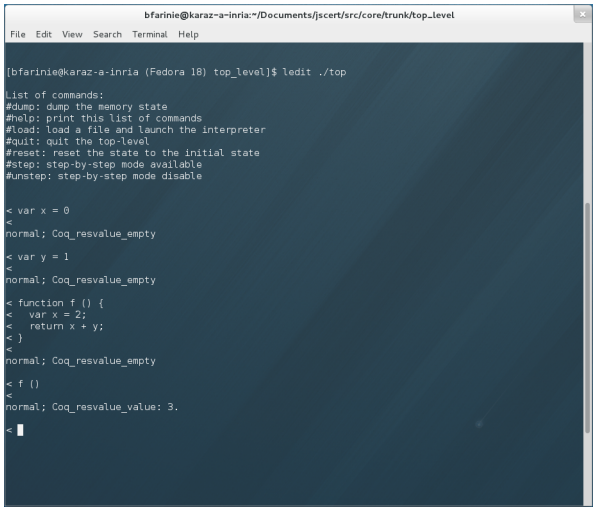
La première partie gère donc l'évolution de l'état et du contexte d'exécution au fil des exécutions successives. Elle est implémentée par un object OCAML contenant les méthodes nécessaires à cette gestion. La deuxième partie interagit avec l'utilisateur et appelle les méthodes de l'objet fourni selon l'action effectuée. Elle transcrit aussi les résultats renvoyés par l'interpréteur et les affiche pour l'utilisateur.

Dans ce programme, l'interpréteur JSREF est utilisé comme une boîte noire. Cela rend le top-level relativement robuste quant aux modifications de celui-ci.

4.2 Quelques fonctionnalités

Je détaille ici quelques fonctionnalités du top-level que j'ai implémenté. Un exemple de son utilisation est donné à la figure 6.

FIGURE 6 – Exemple d'utilisation du top-level



Affichage de la mémoire

L'affichage de la mémoire permet d'afficher l'état dans lequel se trouve l'interpréteur à un instant donné. Cela permet entre autre de voir quelles sont les variables déclarées et ce à quoi elles sont assignées. Cela affiche aussi les branchements complexes des chaînes de prototypage des objets en mémoire.

L'affichage est effectué en parcourant l'état récupéré par le top-level à la fin de l'exécution d'une instruction par l'interpréteur JSREF.

Exécution pas à pas

Sans surprise, cette fonctionnalité permet une exécution pas à pas du programme. Entre chaque pas d'exécution, l'état mémoire de l'interpréteur est affiché avant de continuer l'exécution.

Normalement, une telle fonctionnalité n'est pas possible dans un top-level. En effet, l'interpréteur étant utilisé comme une boîte noire, il n'est pas possible d'accéder aux différentes étapes de l'exécution. Une particularité de l'interpréteur JSREF rend cependant possible cette fonctionnalité.

On rappelle que JSREF est implémenté en COQ, un langage qui ne peut admettre des fonctions ne terminant pas. Pour pallier à ce manque, l'interpréteur JSREF prend en argument une valeur de « carburant ». Au cours de l'exécution, cette quantité diminue et le programme s'arrête lorsque qu'elle atteint 0, assurant

ainsi la terminaison. Pour pouvoir simuler des programmes ne terminant pas, cette quantité est fixée à l'entier maximal supporté par l'architecture, valeur généralement suffisante pour épuiser la patience de n'importe quel humain.

Ainsi pour effectuer une exécution pas à pas, le top-level commence par retenir l'état actuel de la mémoire. Puis il lance l'interpréteur JSREF avec une quantité de carburant de 1. Enfin il affiche l'état mémoire, restaure l'état mémoire à l'état enregistré avant de recommencer une nouvelle exécution en incrémentant la quantité de carburant. De la sorte, une seule étape est effectuée lors de la première passe, puis deux lors de la suivante et ainsi de suite.

5 Générateur de tests

Aujourd'hui, de nombreux navigateurs supportent JAVASCRIPT. Cependant, tous ne l'interprètent pas de la même manière. Cela est quelque fois dû à une ambiguïté dans la spécification ECMASCRIPT5, mais ce sont souvent des erreurs d'implémentation de cette même spécification, comme pour les exemples présentés dans la figure 7. Ces erreurs ne sont pas corrigées pour une simple raison : elles ne sont pas testées par la suite de tests officielle, rendant leur détection très difficile. D'où la volonté de vouloir générer des tests plus exhaustifs pour cette spécification.

Plutôt que des tests vérifiant le bon comportement de l'interpréteur sur des cas complexes, on voudrait générer des tests assurant la validité de l'interpréteur sur tous les cas de base. Et c'est ici que l'on apprécie le style d'écriture de la spécification ECMASCRIPT5. En effet, étant écrite dans un style algorithmique, elle peut être représentée comme un arbre dont les nœuds sont des disjonctions de cas dans la spécification. Pour générer des tests sur tous les cas de bases, il suffit donc de développer un générateur couvrant l'intégralité des premiers niveaux de cet arbre. On pourrait qualifier ces tests de « tests de branchement ».

FIGURE 7 – Confrontation des résultats de trois tests sur les navigateurs avec la spécification ECMASCRIPT5

Test 1 :
```
while(true) {
  try{throw "exception"}
  catch(e){"catch"}
  finally {"finally"; break}}
```

Test 2 :
```
try{ "try" }
finally { "finally" }
```

Test 3 :
```
while(true) {
  try{"try"}
  finally {break}}
```

	Test 1	Test 2	Test 3
	`"finally"`	`"finally"`	`"try"`
	`"finally"`	`"finally"`	`"try"`
	`"finally"`	`"finally"`	`"try"`
	`"catch"`	`"try"`	`"try"`
	`"finally"`	`"finally"`	`"try"`
	`"finally"`	`"try"`	empty value

5.1 Fonctionnement du générateur

L'implémentation de mon générateur est donc basée sur plusieurs sous-générateurs, permettant de générer des blocs de programmes vérifiant une des propriétés testées lors des disjonctions de cas dans la spécification. Par exemple, lors de l'évaluation d'un `try` **Block** `finally` **Finally**, si l'évaluation du bloc **Finally** retourne un type `Normal` on renvoie le résultat de l'évaluation du bloc **Block**, sinon on renvoie celui du bloc **Finally**. Un sous-générateur s'occupe donc de produire des blocs dont on sait que l'évaluation est soit normale, soit abrupte.

Le générateur produit des programmes JAVASCRIPT représentés par une structure abstraite d'arbre. Cette structure abstraite est ensuite transcrite en code réel JAVASCRIPT. Elle est aussi évaluée par un interpréteur dans le but d'ajouter en commen-

taire le résultat attendu pour ce test dans le fichier JAVASCRIPT.

5.2 Critiques

On peut se poser la question de la redondance de mon interpréteur avec JSREF. En fait, ces deux interpréteurs visent des buts très différents. JSREF cherche à pouvoir évaluer n'importe quel programme JAVASCRIPT. Pour ce faire, il doit contenir toute la complexité de la spécification ECMASCRIPT5. Mon générateur lui ne doit qu'évaluer de simples tests de branchement. Ces tests sont excessivement simples, voir souvent triviaux. Cela m'a permit d'implémenter l'interpréteur dans un style copiant presque mot pour mot la spécification ECMASCRIPT5. Ainsi, il devient très aisé de s'assurer de sa conformité vis-à-vis de la spécification.

En l'état, l'interpréteur est capable d'évaluer la plupart des instructions et les sous-générateurs peuvent produire des blocs pour la plupart des branchements. Le générateur est actuellement implémenté de façon à produire des tests pour les instructions de type `try` **Block** `finally` **Finally** et `if` **Expression Statement** `else` **Statement**. Il peut être facilement étendu à d'autres instructions à l'aide des sous-générateurs. La grosse lacune du générateur et de l'interpréteur réside dans le non-gestion actuelle des objets. Cependant la structure modulaire du code peut permettre d'ajouter leur support sans de trop grandes difficultés.

Conclusion

Dans ce rapport, j'ai expliqué le fonctionnement global du langage JAVASCRIPT, dont la compréhension représente une part importante de mon stage. J'ai défini les éléments composant la grammaire de JAVASCRIPT, détaillé le comportement des objets et du prototypage et décrit les mécanismes de gestion des environnements et des contextes. La maîtrise de ces différents points m'a permis de mieux appréhender la complexité du langage JAVASCRIPT.

Les contributions que j'ai présentées visent soit à améliorer la fiabilité de JSREF, l'interpréteur JAVASCRIPT développé dans le cadre du projet JSCERT, soit à en faciliter le développement. Au nombre de quatre, elles mêlent insertion à la compilation d'outils préexistants et développement en OCAML de nouveaux outils. Elles ont aussi été une occasion de m'essayer à un travail de développement au sein d'une équipe de recherche.

Pour finir, je tiens à remercier les membres de l'équipe Celtiques et plus particulièrement Alan Schmitt et Martin Bodin pour leur accueil à l'Inria de Rennes.

Références

[1] JSCERT : Certified JAVASCRIPT, jscert.org/

[2] Annotated ECMASCRIPT5, es5.github.io/

[3] Pretty-Big-Step Semantics, charguéraud.org/

[4] Bisect is a code coverage tool, x9c.fr/

[5] Sergio Maffeis, John C. Mitchell & Ankur Taly, *Operational Semantics for* JAVASCRIPT