現代的 JavaScript エンジンの実装

鈴 木 勇 介[†,††]

ECMA262 の最新版 5.1(June 2011) の仕様に忠実な処理系を実装した. そして TC39 で策定される Conformance Suite である Test262 にて完全に仕様に準拠していることを確認した. 更にこの処理系に対して各種最適化を行った.

Building modern JavaScript Engine

YUSUKE SUZUKI[†,††]

We implemented an engine that is fully compliant with ECMA262 5.1 (June 2011) and confirmed it by using Conformance Suite Test262 provided by TC39. And we optimized it.

1. はじめに

ECMAScript は，JavaScript を技術委員会 TC39 (Technical Committee 39)[☆1]が標準化した言語である. その仕様は ECMA262[☆2]として策定されていて最新版は 5.1 である. いくつかのブラウザには V8[☆3], JavaScriptCore[☆4], SpiderMonkey[☆5]といった処理系が実装されている. 各処理系によって, 仕様に対する準拠の度合いが異なり, 仕様の細かい部分ではバグがあることも多い.

そこで ECMA262 5.1 の仕様に忠実に従ったリファレンス実装としての処理系 lv5 を開発した. この処理系は ECMA262 5.1 に完全対応している.

また, 仕様に準拠していることが確認できたため, 処理系に対して最適化を行った.

2. 仕様準拠度の評価

TC39 では, ECMA262 への仕様準拠度を確認するための Test Suite として Test262[☆6]を提供している. 今回はこの Test を仕様準拠度の指標として用いた. この Test は sputniktests[☆7]と ES5 Conformance Suite[☆8]をあわせ, かつバグを修正し, テストケースを追加したものである.

いくつかの主要なブラウザ, 及び我々が開発した処理系 lv5 に対して Test Suite を実行し評価した.

- `Opera 11.51`
- `Safari 5.1.1`
- `GoogleChrome 15.0.824.120`：V8 3.5.10.23
- `Firefox 8.0`
- `lv5`：今回開発した処理系

そして評価結果を表 1 及び図 1 にまとめた.

現在 lv5 では, Test262 の 11029 件のうち 11011 件, 99.83% pass していて一番準拠度が高い.

残り 18 件については Test262 自体のバグであると考えており, Test262 の開発者にレポートを提出して

表 1 Test 結果

	総数	pass	fail	標準準拠度 (%)
Opera	11029	7276	3753	65.97
Safari	11029	10256	773	92.99
GoogleChrome	11029	10611	418	96.21
Firefox	11029	10865	164	98.51
lv5	11029	11011	18	99.83

† サイボウズ・ラボユース
Cybozu Labs Youth
†† 慶應義塾大学理工学部 3 年
3rd grade of Department of Science and Technology, Keio University

☆1 `http://www.ecma-international.org/memento/TC39.htm`
☆2 `http://www.ecma-international.org/publications/standards/Ecma-262.htm`
☆3 `http://code.google.com/p/v8/`
☆4 `http://www.webkit.org/projects/javascript/`
☆5 `https://developer.mozilla.org/en/SpiderMonkey`
☆6 `http://test262.ecmascript.org/`
☆7 `http://code.google.com/p/sputniktests/`
☆8 `http://es5conform.codeplex.com/`

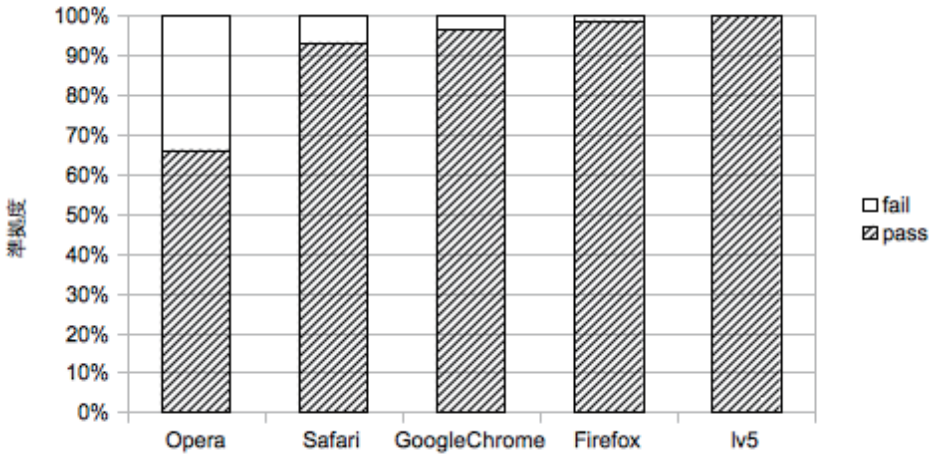

図 1 標準準拠度の評価

いる．☆1☆2☆3☆4 よって，それを考慮すると Test262 のテストに全て通っていると考えている．

3. 最 適 化

開発した lv5 は当初の目標であった仕様準拠度を達成したので，次は lv5 の最適化に着手した．以下，実装した最適化手法について解説する．

3.1 NaN boxing

ECMAScript は動的型言語であり値に型のタグを付ける必要がある．このタグを含めた値を以下 JSVal と定義する．JSVal ではタグとその値を別にとる必要があり，もしも浮動小数点数を値にそのまま格納する場合，本来 JSVal は 64bit より大きなサイズとなる．

ここで図 2 に示す IEEE 754 の倍精度浮動小数点数の形式を利用する．IEEE 754 では指数部 (Exp) が全て 1 かつ仮数部 (Fraction) が 0 以外の場合 quiet NaN であると定義されており，NaN には多くの領域が割り当てられている．この時 NaN をある一種類に正規化することで，他の NaN を浮動小数点数以外の別の値を表すものとして利用し，例えばポインタ値などを boxing することができる．これは LuaJIT☆5 で用いられた手法で，JavaScriptCore や SpiderMonkey にも実装された．

図 2 IEEE 754 倍精度浮動小数点数の形式とビット幅

この時 32bit 環境では，上位 32bit をタグに使うことで下位 32bit に好きな値を入れることができる．これによってポインタ，int32_t を boxing することができ，JSVal の値を 64bit に収めることができる．

一方 64bit 環境ではポインタの大きさが 64bit である．しかし，Linux, Windows などの OS ではポインタの上位 16bit が常に 0 であり，以下の方法で boxing が可能である．ポインタがそのような性質を持たない Solaris などでは JSVal は 128bit となる．☆6

NaN を正規化することで上位 16bit は 0x0000 から 0xFFF0 の範囲内に収まる．この最大値は-Infinity の時である．ここで上位 16bit に 1 を加えることで，この範囲を 0x0001 から 0xFFF1 にずらす．結果，0x0000 をポインタ用のタグとして利用でき，ポインタを変更することなく格納することが出来る．

正規化された以外の NaN のビットパターンをポインタのタグと定めれば☆7範囲をずらすことなくポインタを格納することができる．ただし，ポインタの上位 16bit は 0x0000 であることから，この場合はポインタの値を変更する必要があり，保守的 GC からたどることができなくなってしまう．

lv5 では，利用している BoehmGC☆8が保守的 GC であるため，ポインタの値を変更せずに格納する手法を採用している．

☆1 https://bugs.ecmascript.org/show_bug.cgi?id=215
☆2 https://bugs.ecmascript.org/show_bug.cgi?id=216
☆3 https://bugs.ecmascript.org/show_bug.cgi?id=217
☆4 https://bugs.ecmascript.org/show_bug.cgi?id=218
☆5 http://luajit.org/

☆6 https://bugzilla.mozilla.org/show_bug.cgi?id=577056
☆7 例えば 0xFFF1
☆8 http://www.hpl.hp.com/personal/Hans_Boehm/gc/

3.2 Stack VM の実装

初期の実装は単純な AST インタープリタであり, ECMA262 の説明用アルゴリズムに非常に近い実装であった. しかし速度上非常に問題があったので, スタック VM を実装した.

この VM では制御フレーム用, 変数用などを区別することなく単一の配列としてスタックを構築した.

また, コンパイル時点での関数が必要とするスタックの大きさを解析し, 関数呼び出しの際にスタックの上限を超えないかを検査する. もし超えていた場合は ECMAScript の例外を投げることでスタックオーバーフローを回避した.

VM が ECMAScript の関数を呼び出した場合, フレームをスタックに積んでジャンプするだけである. このため, ECMAScript の関数を呼び続けている間は VM のメインループの外に出ることはなく, また C スタックも新たに消費することはない. また, VM メインループから外れるのは ECMAScript の例外が起こった時のみで, finally や return といった制御構造は全てメインループ内におさまる.

3.3 ECMAScript における変数解析

変数とその参照を解析し, STACK, HEAP, DYNAMIC, GLOBAL の 4 種に分類し, それぞれに対して異なる最適化を行った.

ある関数が利用する変数は環境と呼ばれる領域に割り当てられる. 環境は通常ハッシュテーブルを使って実装し, クロージャに対応するためにヒープ上に構築される. 通常, 局所変数も環境に割り当てられる. しかし, ある変数がその関数の局所変数として宣言され, かつその関数の中からのみ参照される場合, 関数終了後この環境の変数は変更されることはない. したがってヒープではなくスタックに割り当てる最適化をすることができる. そのような変数を STACK 変数と分類した.

一方, ネストした別の関数の中から参照される場合がある. この場合環境の変数は, 環境と共に変更され得るため, ヒープ上に割り付けなければならない. これを HEAP 変数と分類した. ただし, この時, ある識別子がどの環境の HEAP 変数を指しているかは静的に解析可能である. このため, 事前にその環境中の変数のアドレスが決定し, ハッシュテーブルを引かずにすむ最適化を行える.

また, トップレベルの環境が Global オブジェクトである評価環境の場合, 参照先変数が見つからないならそれは Global オブジェクトへの変数参照である. このため, 直接 Global オブジェクトのプロパティを辞書引きすることで参照を最適化できる. このような変数を GLOBAL 変数と分類する.

```
function stack() {
  var i = 20;
  print(i);
}

function heap() {
  var i = 20;
  function inner() {
    print(i);
  }
  inner();
}

var i = 20;
function global() {
  print(i);
}

function with_dynamic() {
  var obj = {};
  with (obj) {
    print(i);
  }
}

function with_not_dynamic() {
  var obj = {};
  with (obj) {
    var i = 20;
    function inner() {
      print(i);
    }
  }
}

function eval_dynamic() {
  eval("var i = 20");
  print(i);
}
```

図 3 変数の種類

ECMAScript には環境に影響を与える幾つかの要素が存在し, これらが現れた場合, 静的に解析することができない. そのような参照を DYNAMIC と分類する. この場合環境を一つ一つたどって変数を参照することになる.

(1) **with**

図 3 の with_dynamic 関数の変数 i の参照のように

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

Point.prototype.add = function(rhs) {
  return new Point(
    this.x + rhs.x,
    this.y + rhs.y);
}

var point = new Point(0, 10);
```

図 4 class 的利用の例

with の中の識別子が with の中で対象の変数領域を見つけることが出来なかった場合, この識別子は with に指定されたオブジェクトに trap されうる. この場合は DYNAMIC と分類する必要がある. 一方で with_not_dynamic 関数の i のように内部で trap された場合, 動的に行う必要はない.

(2) **direct call to eval**

eval 関数が eval という識別子で呼ばれた場合, direct call to eval という. この時, eval の評価環境は, eval が記述されている関数の環境, もしくは strict モードの場合は, そこから新たに 1 層環境を作成し, その環境にて評価される. この時, eval 内でどのような変数の参照されるかどうかは不明なため, eval の存在する環境より上にある環境に存在する変数は, すべてヒープ上に割り付けなければならない. 図 3 の eval_dynamic 関数の変数 i の参照は DYNAMIC である.

(3) **arguments**

arguments は環境と連携して機能し, 環境の変数を変更しうる. よって strict モードでない場合, arguments の識別子の現れた関数の引数は, HEAP 変数にしなければならない. これについては後述する.

3.4 Inline Cache

Inline Cache は Self にて開発された手法で[☆1], V8 に Hidden Class[☆2]として実装されたものである. これは V8 において良いスコアを示し, JavaScriptCore にも採用された[☆3].

ECMAScript は, prototypal な言語であり, 構文としてはっきりとしたクラスが存在するわけではない. クラスは比較的静的に確定することが期待され, プロパティの参照についてキャッシュを取るということができるが, ECMAScript においてはクラスがはっきりとしないため, キャッシュを取るべきターゲットを決定することが難しい.

しかし, 現実の ECMAScript のプログラムはある程度クラスのようなものを想定している. あるオブジェクトにおいては決まった名前のプロパティが追加される.

図 4 の場合, Point コンストラクタから作られたオブジェクトは x, y という名前のプロパティにアクセスされる可能性が高い. また, add メソッドに渡される rhs は Point のコンストラクタから生成されたオブジェクトである可能性が高い. このような暗黙的なクラスを V8 では HiddenClass[☆4], JavaScriptCore では Structure という.

ここで, ある同一の Map に対し同じ名前のプロパティが同じ順番で追加された Map は同じ Map になるように操作を行う. プロパティが追加されたときに新しい Map を作り, 前の Map に追加されたプロパティのシンボルと生成した Map を記録しておき, transit を行う. 以降, 同じ Map から同じシンボルが追加されて transit した場合, この記録を検索し, すでに transit した Map があればそれを利用する. これによって同じ Map から同じ順番で同じシンボルのプロパティが追加された Map は同じ Map となる.

そして, rhs.x といったプロパティの参照のバイトコード上に, 以前にプロパティを参照した結果の Map とプロパティ位置のオフセットをキャッシュする. 具体的には機械語もしくはバイトコード上に即値として書き込み, 動的にコードを変更する.

結果として, このバイトコード上で全く同じ Map をもつオブジェクトのプロパティの参照が行われた場合, キャッシュした Map との比較を行い, 一致すれば高速なオフセットによるアクセスを行うことができる.

3.5 Global 変数の参照

ECMAScript では Global な環境は Global オブジェクトとして言語上露出している. これはプロパティを設定するとトップレベルの変数が増えるということであり, 結果として Global 変数は静的に判断することができない.

そこで, Global オブジェクトも ECMAScript のオブジェクトであることから, Global 変数の参照に対しても前述の Inline Cache を行う. 結果として高速な

☆1 http://labs.oracle.com/self/papers/pics.html
☆2 http://code.google.com/intl/ja/apis/v8/design.html#prop_access
☆3 http://www.webkit.org/blog/214/introducing-squirrelfish-extreme/
☆4 ソースコード上は Map

アクセスを行うことができる.

lv5 では, これは Own Property に限って行っている. これは Global 環境の変数を参照する場合その参照方式は識別子によるもので, その [[Prototype]] の Object.prototype のプロパティを参照することはほぼないと考えられるためである.

3.6 不要環境の作成排除

すべての変数がスタック上に収まり, かつ eval が対象の関数内になく新たに変数を作られることもない時, 環境に変数が追加されることはない. この時, 環境を作らないようにする最適化を行うことができる. これは環境の生成と破棄のコスト, 参照した時のネストの深さを下げることができ, 特にスタック変数しか利用しない小規模な関数を大量に呼び出す場合に有効である.

3.7 String の Rope 表現

ECMAScript の String は不変であることが仕様上保証されている. このため, 多くの処理系は String を Rope で実装しており, lv5 もそれに倣い Rope による String の実装を行った.

3.8 デッドコードの削除

デッドコードを判定, 削除した.

3.9 バイトコードベースの RegExp の実装

以前は V8 と JavaScriptCore が過去利用していた JSCRE を利用していたが, ECMA262 の一部仕様に準拠していないこと及び速度が非常に遅いことから, バイトコードベースな RegExp を実装した.

3.10 strict モード下の最適化

ECMA262 では 5 より新たに strict モードという機能が追加された. strict モードは静的に最適化阻害要因を抑制し, 結果として strict モード下では幾つかの最適化を行うことができる.

3.10.1 arguments の環境連携の削除

arguments は strict モード下では環境の値が変更されても値が変更されない. 図 5 の normal 関数について, arguments[0] に代入が行われると, それに紐づいた引数 x の値も変動する. よって, 従来, arguments という識別子が文法上現れると, パラメータは全て HEAP 変数にする必要があった. これは arguments が環境と連携しているためで, arguments を介して変数が変更される可能性があり, arguments を外に持ち出すことが可能なためである.

しかし, strict モード下では arguments は環境と連携していない. よって, strict 関数においては lv5 は x を STACK 変数として最適化することができる. この結果, 外部関数から参照される変数がなくなり, STACK 変数だけとなるため, HEAP に環境を作らない最適化も行われる.

```
function normal(x) {
  arguments[0] = 20;
  console.assert(x === 20);
}
normal(10);

function strict(x) {
  "use strict";
  arguments[0] = 20;
  console.assert(x === 10);
}
strict(10);
```

図 5 strict モード下の arguments

```
function normal() {
  var i = 20;
  function outer(str) {
    eval(str);
    function inner() {
      print(i);
    }
  }
}

function strict() {
  "use strict";
  var i = 20;
  function outer(str) {
    eval(str);
    function inner() {
      print(i);
    }
  }
}
```

図 6 通常 mode 及び strict モード下の eval

3.10.2 eval 最適化

strict モードでの eval は変数のスコープが分離されるので, 呼び出し元の環境の変数は影響を受けない.

図 6 の関数 normal について, もしも outer("var i = 50") と呼び出されると eval によって新たに作られた変数 i に trap されるため, 静的に変数 i の位置を紐付けることはできない. この場合は環境を動的に辞書引きする参照となる.

ところが strict モードが有効な場合, 図 6 の関数 strict については, inner の変数 i の参照が outer の eval によって trap されることはない. これは strict

表 2 SunSpider 結果

	所要時間 (ms)
lv5 最適化前	55000
lv5 最適化後	1830
Opera	212
Safari	185
GoogleChrome	214
Firefox	223
SpiderMonkey JIT 無効	950
SpiderMonkey C	3657

モード下では eval は新たに環境を 1 層作り, そこで評価を行うため, 呼び出し元環境に変数を追加するといったことがないためである. このためこの例では i の参照は一意に strict 関数の局所変数 i に紐付くことが解析可能であり, lv5 では通常の HEAP 参照に最適化される.

4. 最適化の評価

処理性能の評価として, 一般的に利用されている SunSpider☆1を用いた. これは, あるインターフェースに従ったプログラムを端末から容易に実行できる機能を持っている. 評価結果を表 2 にまとめた. 主要処理系に比べて約 10 倍遅いが, 最適化する前に比べると約 20 倍の高速化を達成した.

5. 将来的な課題

将来的に以下の機能を実装することを予定している.

5.1 自作 GC の実装

現在は利用している BoehmGC は汎用 GC のため, ファイナライザが個々に登録されるなど一部コストが高いという問題がある. そのため新たに GC を実装し, 置き換えることを検討している.

実装としては, C++の RAII を利用した明示的なスコープの管理による Exact GC で, 単純な Mark & Sweep 方式を想定している.

5.2 JIT

近代的な ECMAScript 処理系は全て JIT エンジンを搭載している. JIT エンジンを導入することによって dispatch の高速化が期待できる.

現在は, V8 が採用するスタック VM と JavaScriptCore が採用するレジスタ VM とどちらの方式の機械語を出力すべきか検討している. レジスタ VM を採用する場合, 現在の VM を変更する必要がある. なお, JIT の場合は書き換え可能なプロパティ参照コードを高速ケースとして設置し, のちに書き換えることによって Inline Cache を実現する.

5.3 RegExp JIT

現在 RegExp はバイトコードベースであるが, これを機械語に変換して実行する. 現在の時点で RegExp VM はかなり簡素なため, バイトコードに当たるものをそのまま機械語に置き換えるコンパイラの実装を検討している.

5.4 V8 互換 API

V8 は, 組み込み ECMAScript 処理系として node.js☆2で採用されるなど, 代表的な存在となっており, V8Monkey☆3といった SpiderMonkey に V8 の API を実装しようというプロジェクトも存在する. このため, lv5 でも V8 の互換の外部 API を実装することを検討している.

6. ま と め

Test262 の結果を指標として, ECMA262 5.1 に完全対応する処理系, lv5 を開発した. lv5 はオープンソースで開発している.☆4

今後の課題として, より実用的な速度で動作するよう最適化を行うことを検討している.

謝辞 本実装について全面的に開発支援を行ってくれ たサイボウズ・ラボユース及びサイボウズ・ラボのメ ンバー, 特に竹迫良範氏と光成滋生氏, 西尾泰和氏, 蓑輪太郎氏, 中谷秀洋氏に感謝する.

参 考 文 献

1) Urs Hlzle, Craig Chambers, and David Ungar: Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches (1991).
2) Russ Cox: Regular Expression Matching: the Virtual Machine Approach (2009).
3) Standard ECMA-262 ECMAScript Language Specification Edition 5.1: http://www.ecma-international.org/publications/standards/Ecma-262.htm.
4) V8: http://code.google.com/p/v8/.
5) WebKit: http://www.webkit.org/.

☆1 http://www.webkit.org/perf/sunspider/sunspider.html

☆2 http://nodejs.org/

☆3 https://github.com/zpao/v8monkey

☆4 https://github.com/Constellation/iv