平成 30 年度

修士学位論文

対象のアプリケーションが使う データ型に特化した JavaScript VM 開発フレームワーク

A Framework for Development of JavaScript VMs Specialized for Datatypes That Target Application Uses

1215079　　片岡 崇史

指導教員　　鵜川 始陽

2019 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

対象のアプリケーションが使う
データ型に特化した
JavaScript VM 開発フレームワーク

片岡 崇史

JavaScript は高い生産性を提供するプログラミング言語で，JavaScript のイベント駆動的なプログラミングスタイルは組込みシステム開発と親和性が高い．しかし，組込みシステムは一般的に計算資源が乏しいため，大きな仮想機械（以下，VM）を必要とする JavaScript を実行することは難しい．そこで我々は，実行するアプリケーションに適した VM を作ることによって，VM のサイズを小さく，また高速にするというアプローチをとる．本研究では，アプリケーションの性質に合わせてカスタマイズした JavaScript VM を生成するためのフレームワークを開発する．このフレームワークは，アプリケーションが使用するデータ型に注目した二つのカスタマイズを行う．一つは，VM 内部で頻繁に行われる型ディスパッチ処理を高速化するために，VM 内部のデータ型の表現を対象のアプリケーションに特化することである．もう一つは，生成する VM のサイズを小さくするために，対象のアプリケーションが使うことのないデータ型を処理する機能を，生成する VM から省くことである．さらに，アプリケーションの実行に必要なデータ型を特定するためのプロファイラも開発する．このプロファイラを利用することによって，不要なインタプリタの機能を省いた VM を比較的簡単に作ることができる．評価の結果，eJSTK がカスタマイズした VM のインタプリタは，サイズが半分ほどに小さくなることが分かった．また，eJSTK を使って適切にカスタマイズされた VM は，手動でチューニングされた VM と比較して実行が高速であった．

キーワード　JavaScript，仮想機械，組込みシステム，プロファイラ，ユニットテスト

Abstract

A Framework for Development of JavaScript VMs Specialized for Datatypes That Target Application Uses

Takafumi Kataoka

JavaScript is a programming language that offers high productivity, and JavaScript's event driven programming style matches development embedded systems well. However, it is difficult to execute JavaScript programs, which require a large virtual machine (VM), on embedded systems because an embedded system typically has poor computing resources. Therefore, we take an approach to generate an appropriate JavaScript VM to a specific application that is small and runs fast In this research, we develop a framework to generate a JavaScript VM that is customized for an application. The framework performs two customizations focusing on the datatypes that the application uses. Firstly, eJSTK specializes the representations of internal datatypes of a VM to a target application to increase execution speed of type based dispatching, which is frequently executed in a JavaScript VM. Secondly, eJSTK omits the functionalities of processing those datatypes from the generated VM that the target application never uses to keep the generated VM small. Furthermore, we develop a profiler to identify necessary datatypes for executing the target application. By using this profiler, the users can easily develop a VM from which unnecessary functionalities are omitted. As a result of our evaluation, we found that eJSTK reduced the size of the interpreter of the customized VM by half. In addition, properly customized VMs

were faster than manually tuned VM.

key words English JavaScript, virtual machine, embedded system, profiler, unit testing

目次

目次

図目次

表目次

第 1 章

はじめに

近年，組込みシステムの開発 [1] が盛んである．一般的に組込みシステムに使われる機器は，計算資源が乏しい．そのため，組込みシステムの開発には，C 言語やアセンブリ言語などといった，高速に動作し，またメモリを多く必要としないプログラミング言語が利用されることが多い．

しかし，これらの言語を使ったアプリケーション開発には時間がかかり，プロトタイプ開発には向いていないなど，生産性に問題がある．また，自動メモリ管理システムを持たないため，メモリ管理はプログラマ自身の責任で行わなければならない．

この問題を解決するために，本研究では JavaScript で組込みシステムの開発を可能とすることを目指す．JavaScript は，C 言語やアセンブリ言語と比較して開発を速く行うことを可能とする．仮想機械（以下，VM）上で動作し，メモリ管理は自動的に行われるため，プログラマがメモリリークの心配をする必要はない．また，JavaScript のイベント駆動的なプログラミングスタイルは，組込みシステムの開発と親和性が高い [2, 3].

しかし，計算資源の乏しい機器の上で JavaScript を動作させるためには，VM はサイズが小さく，動作が高速でなければいけない．JIT コンパイル [4] のような，動的な最適化は大きなメモリ領域を必要とするため，組込みシステムに使われるようなメモリに制限のある環境には適していない．そこで我々は，実行する対象のアプリケーションに VM を特化させるというアプローチをとる．以下では，アプリケーションに合わせて特殊化した VM を，**eJSVM** と呼ぶ．

JavaScript は，同じ演算子でもオペランドに与えるデータ型が異なれば，実行する処理が異なる．例えば加算演算子「+」は，1+2 のように両オペランドに数値が与えられた場合は

数値の加算として処理する．一方，1+"2"のように，オペランドに数値と文字列が与えられた場合，数値を文字列に変換してから文字列同士の連結を行う．このため，JavaScript VM は演算子を処理するとき，オペランドに与えられたデータ型を判別し，それによって実行する処理を振り分けることが必要になる．このようにして処理の振り分けることを型ディスパッチと呼ぶ．また型ディスパッチを行う機構を型ディスパッチャと呼ぶ．

eJSVM は，値に付属する二種類の動的型情報によって値のデータ型を表現する．型ディスパッチャはこの型情報によって値のデータ型を判別する．動的型情報の一つは，オブジェクトを指すポインタの下位 3 ビットで表現されるポインタタグである．もう一つは，ヘッダタグである．ヘッダタグは，ポインタが指すオブジェクトのヘッダに含まれる．ポインタタグはオブジェクトにアクセスせずに取得することができるため，ポインタタグのみで判定できるデータ型は高速に扱うことができる．しかし，ポインタタグには 3 ビットしか使えないため，表現できるデータ型の数が少なく，eJSVM で扱うすべてのデータ型を表現することができない．ここで本研究では，次の二つのアプリケーションの特性に注目し，VM を特殊化する．

一つは，アプリケーションが頻繁に使用するデータ型である．例えば，ある数値計算を多く行うアプリケーションでは，演算子のオペランドとして数値が頻繁に与えられるとする．そのようなアプリケーションを，数値を高速に扱うことのできる VM の上で実行できれば，アプリケーション全体の実行速度が速くなることが期待できる．本研究では，アプリケーションが頻繁に扱うデータ型をポインタタグで表現することで，そのデータ型を高速に扱うことができるようにした VM を作る．

もう一つは，アプリケーションの実行に必要な演算子の機能である．アプリケーションによって演算子のオペランドに与えられ得るデータ型は異なる．例えば，あるアプリケーションの実行において，1+2 というように加算演算子の両オペランドに数値が与えられることはあっても，1+"2"のように，数値と文字列が与えられることはない場合がある．そのようなアプリケーションを実行する VM に，数値と文字列がオペランドに与えられた場合の加算演算子の処理コードは必要ない．我々は，このようなコードを VM から除くことによって，

VM のサイズを小さくする．

本研究では，以上で述べたことに注目し，特定のアプリケーションに合わせて VM を特化するフレームワーク eJSToolKit （以下，**eJSTK** という）を開発する．このフレームワークは，対象とするアプリケーションのの特徴を表現した二つのパラメタを入力として受け取る．一つは，各データ型について，ポインタタグとヘッダタグのどちらで表現するかを示したものである．もう一つは，各演算子について，オペランドにどのデータ型が与えられ得るかを示したものである．

さらに本研究では，アプリケーションの実行において演算子のオペランドに与えられ得るデータ型を特定するためのプロファイラも開発する．これをを用いることで，ユーザは eJSTK への入力パラメタの一つを自動的に決めることができる．このプロファイラは，アプリケーションのユニットテストの実行トレースを利用する．アプリケーションのユニットテストをそのまま利用して演算子のオペランドに与えられ得るデータ型を特定できるため，ユーザ自身が把握するよりも，簡単にコンパクトな VM を作ることができる．

本研究の貢献は以下の通りである．

- アプリケーションが使用するデータ型に注目して特化した VM を作るフレームワークを開発した．
- アプリケーションが使用するデータ型に注目した VM の特殊化が，VM サイズと実行速度に与える影響を明らかにした．
- ユニットテストを用いて演算子のオペランドに与えられ得るデータ型を特定するプロファイラを作成した．
- 演算子のオペランドに与えられ得るデータ型を特定するために，ユニットテストを利用することの妥当性を示した．

本論文の 3 章と 4 章，5 章は論文 [5] で，7 章の 1 節は 論文 [6] で発表したものである．

第 2 章

関連研究

2.1 インタプリタの自動生成に関する研究

Vmgen [7, 8] は，スタックマシンの計算モデルで動作する VM の自動生成ツールである．Vmgen は，VM 命令の動作定義を入力として受け取り，C 言語の VM 命令コードを生成する．Vmgen と eJSTK はどちらもインタプリタのコードを生成するが，二つの大きな違いがある．一つ目は，カスタマイズの方針が異なるところである．Vmgen は，ユーザが必要な VM 命令を定義すれば，様々なプログラミング言語の VM を生成できるという意味で「カスタマイズ可能」である．Vmgen を用いた言語の実装として，Gforth [9] や Cacao JVM [10] がある．それに対して eJSTK の対象の言語は JavaScript のみである．eJSTK は，実行するアプリケーションが使用するデータ型に合わせた VM を作るために VM 内部をカスタマイズする．

二つ目は，Vmgen が静的型のプログラミング言語を対象とするのに対して，eJSTK は動的型の JavaScript を対象としている．Vmgen の命令定義の記述は，スタックから出し入れする値に型がついていることを仮定している．例えば，Java VM の `iadd` 命令は，二つの整数をスタックから取り出し，計算結果の整数をスタックに入れる．このように，各命令が扱うデータ型は，対象のプログラミング言語によって決められていることを想定している．これと比較して，eJSTK が対象とする言語は JavaScript であり，各 VM 命令が扱う値のデータ型は実行時に決定する．従って，各命令は実行時に型ディスパッチを行う必要がある．

2.2　組込みシステム向けの言語実装

Beatty ら [11] は，組込みシステム向けの効率的な Java の VM を提案している．この VM では，Java アプリケーションのバイトコードは，様々な最適化が行われる中でスレッデッドコードに変換される．これにより命令ディスパッチの実行コストが削減される．それに対して，eJSTK は各命令で実行される型ディスパッチの実行コストを削減することに焦点をあてる．

mruby [12, 13] は，主に組込みアプリケーションを対象とした，Ruby の軽量な実装である．mruby と eJSTK は目的が同じであるが，アプローチが異なる．mruby は，モジュール単位で必要な機能を選択するのに対し，eJSTK は必要なデータ型を選択し，その表現をカスタマイズできるようにした．

2.3　プログラム実行時の最適化

インタプリタの動的なカスタマイズや最適化を行う研究が多くなされている．Quickening [14, 15] や superinstruction [16, 17] はよく知られた最適化手法である．Quickening は，主に命令のオペランドに関する実行時情報に基づいて VM 命令を特化したものに置き換える手法である．Superinstruction は，頻繁に連続して実行される複数の命令を，一つの命令に併合したものに置き換える手法である．eJSVM ではこれらの手法を適用していないが，今後適用することができると考えている．

Würthinger ら [18] と Humer ら [19] は，プログラム実行中に抽象構文木の上で最適化を行う AST インタプリタを提案している．これは，Java で記述されたゲスト VM（AST インタプリタ）を，C++で記述されたホスト VM 上で実行するというように，階層的なアプローチをとっている．この AST インタプリタは，AST の部分木をより高速に実行できる AST に置き換える．この手法は，上で説明したような VM 命令を書き換える最適化よりも汎用的だが，計算資源が制限されている組込みシステムに適用することは難しい．

2.4　動的プログラミング言語における型情報の収集

動的プログラミング言語の型情報を静的に調べることは難しい．多くの動的プログラミング言語の処理系は，プログラムの実行中に現れたデータ型を収集し，投機的な最適化を実行時に行う [20, 21]．しかし，本研究では VM のサイズを小さくすることが目標である．そのため，実際にアプリケーションを実行する前に型情報を収集する必要がある．

Haupt ら [22] は，ユニットテストの実行から型情報を収集する手法を提案している．本手法と同様に，この手法は網羅率の高いテストを利用者に要求するが，複雑で型解析が難しいプログラムであっても，非常に高い精度で実行時に扱う型を特定できることが示されている．Haupt らの手法はプログラムを高速に実行することを目標とするが，本研究で開発するプロファイラの目的は，アプリケーションの実行に必要のない機能を見つけることによって VM を小さくすることであり，本研究ではその効果を検証する．

Serrano [23] は，基本的な型解析だけでは型が正確にわからない変数の出現に関しては，その変数の使われ方から型を推論し，コードを特殊化する手法を提案している．例えば，ある関数の仮引数 `a` があり，関数内では `a.length` というように `a` の `length` プロパティにアクセスしていたり，整数であることが保証されている変数 `i` がある中で `a[i] = x` というようなプロパティへの代入があったとき，高い確率で `a` は配列であることを想定して書かれたプログラムであることが考えられる．このような型に関するヒントをプログラムから得て，それを基に投機的に特殊化したコードを生成する．eJSTK は，VM 命令のオペランドに与えられることのない内部データ型に対応するコードを削除するため，与えられることの決してない内部データ型を多く見つけることが重要であるが，Serrano らによる手法ではそれはできない．

2.5　記号実行によるテスト入力値の自動生成

記号実行系を使ってテスト入力値を自動生成するための研究がなされている．記号実行では，プログラムを解析し，あらゆるパスに対してパスを通る条件を抽出し，その条件を満た

すような入力値を SMT ソルバを利用して生成する．これによって，コード網羅率の高いテスト入力が生成できる．

JavaScript の記号実行エンジンには，Kudzu [24]，SymJS [25]，Jalangi [26] がある．Kudzu は，アプリケーションのセキュリティ上の問題を見つけることを目的としている．SymJS と Jalangi は，網羅率の高いテスト入力を生成することができる．また，記号実行エンジンを用いたユニットテストを自動生成する研究もされている．谷田ら [27] は，JSDoc[*1] の形式で記述された，関数の引数や返り値の型を利用して，ユニットテストに必要なスタブやドライバを自動生成する．

本研究で開発するプロファイラは，網羅率の高いテストをプログラマに要求する．この負担を軽減する方法として，このような技術の利用を検討することは今後の課題である．

[*1] http://usejsdoc.org/

第 3 章

eJSVM

3.1 JavaScript の言語仕様

JavaScript は，プロトタイプベースのオブジェクト指向を特徴としたスクリプト言語である．よく知られている JavaScript 実行エンジンに，V8 [28] や Rhino [29]，SpiderMonkey [30] などがある．

表 3.1 に，JavaScript のデータ型を示す．言語仕様 [31] で定められているデータ型は，表の一番左の列に示している．この 6 つのデータ型のうち，Object のみがプロパティを持つ．Object は，さらに通常のオブジェクトである Object Object や配列オブジェクト Array，関数オブジェクト Function などに分類することができる．そのため我々は，実際の JavaScript のデータ型は表の 3 列目に列挙したものであると考える．以下では，これらの型を **JavaScript のデータ型**と呼ぶ．また，JavaScript のデータ型の実装として，VM 内部で扱うデータ型を**内部データ型**と呼ぶ．eJSVM が用いる内部データ型については，第 3.2 節で説明する．

JavaScript は動的な言語であることから，プログラムの実行中にある値のデータ型に基づいて型ディスパッチを行ったり，あるデータ型の値を別のデータ型の値に型変換することが多い．これは，一部の演算子は，オペランドに与えられたデータ型によって実行する処理が異なったり，オペランドに与えられた値を期待するデータ型に型変換することがあるためである．

例として，加算演算子「+」を考える．加算演算子は，両オペランドに数値が与えられたとき，それらを足した値を返す．一方，どちらかのオペランドに文字列，もしくは文字列に

表 3.1 JavaScript のデータ型

仕様上の型	オブジェクトの分類	JavaScript のデータ型	説明
Undefined		Undefined	未定義値
Null		Null	ヌル値
Boolean		Boolean	真偽値
String		String	文字列
Number		Number	数値
Object	Object Object	SimpleObject	通常のオブジェクト
	Array	Array	配列オブジェクト
	Function	Function	関数オブジェクト
	Regexp	Regexp	正規表現オブジェクト
	Boolean Object	BooleanObject	真偽値オブジェクト
	String Object	StringObject	文字列オブジェクト
	Number Object	NumberObject	数値オブジェクト

変換することができるオブジェクトが与えられた場合，もう一方のオペランドを文字列に変換してから，二つの文字列を結合する．そうでない場合は，両オペランドを数値に変換し，足した値を返す．このような処理の振り分けはプログラムの実行中に頻繁に起こる．そのため，JavaScript VM は与えられた値のデータ型の判別と，それに基づく型ディスパッチを効率的に行うことが重要である．

3.2 eJSVM の内部構造

3.2.1 概要

eJSVM は，組込みシステム上で実行する JavaScript プログラムのための，レジスタベースの VM である．JavaScript プログラムを専用のコンパイラ（以下，eJS コンパイラ）で

表 3.2　JavaScript のデータ型と内部データ型の対応

JavaScript のデータ型	内部データ型	説明
Boolean	special	真偽値
String	string	文字列
Number	fixnum	61 ビット符号付整数
	flonum	倍精度浮動小数点数
SimpleObject	simple_object	通常のオブジェクト
Array	array	配列
Function	function	ユーザ定義関数
	builtin	組込関数
Regexp	regexp	正規表現オブジェクト

コンパイルして得られる VM 命令列のインタプリタとして動作する．現在の eJS コンパイラは，ECMAScript 5.1 [31] で定義されている JavaScript の仕様のうち，関数 `eval()` のような複雑な実装を必要とする機能を除いた仕様をサポートする．eJSVM のメインループは，eJS コンパイラから得た VM 命令を，スレッデッドコードの手法 [32] に基づいて順に実行する．

eJSVM の命令セットは，我々が設計したものを使う．命令実行の例として，加算演算子「+」の処理に対応する VM 命令である add 命令を説明する．この命令は，三つのオペランドをとる．一つ目は，結果を格納するレジスタである．二つ目と三つ目は入力のレジスタであり，加算演算子のそれぞれ左オペランドと右オペランドに対応した値を保持する．この命令の実行には，VM はまず二つの入力レジスタが保持する値のデータ型を判別し，そのデータ型の組に対応した処理に制御を移す．59 個ある VM 命令のうち 26 個の VM 命令は，実行時にこのような型ディスパッチを行う必要がある．

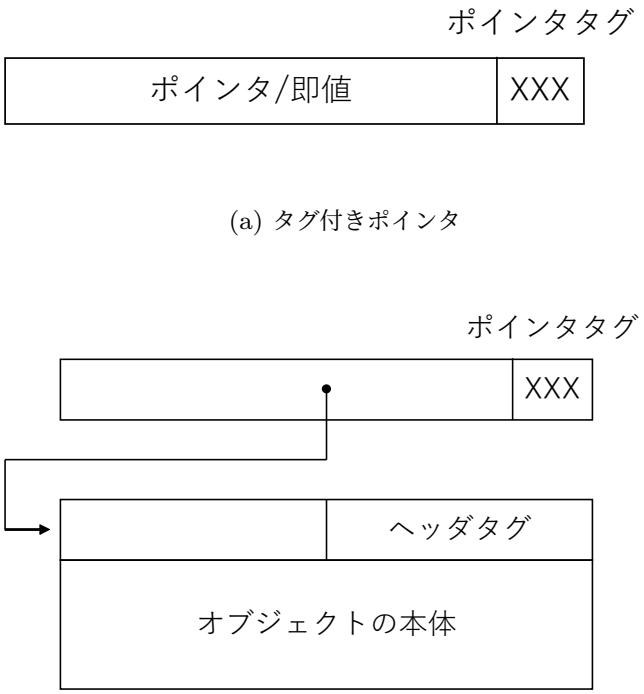
表 3.3　内部データ型とデフォルトのデータ型表現

内部データ型	データ型表現	ポインタタグ	ヘッダタグ	ポインタ/即値
special	normal_special	110	—	即値
string	normal_string	100	4	ポインタ
fixnum	normal_fixnum	111	—	即値
flonum	normal_flonum	101	5	ポインタ
simple_object	normal_simple_object	000	6	ポインタ
array	normal_array	000	7	ポインタ
function	normal_function	000	8	ポインタ
builtin	normal_builtin	000	9	ポインタ
regexp	normal_regexp	000	11	ポインタ

3.2.2　データ型表現

eJSTK では，JavaScript のデータ型に対応する形で，eJSVM が扱う内部データ型を定義している．表 3.2 に，eJSVM における内部データ型と，JavaScript のデータ型との対応関係を示す．JavaScript のデータ型と内部データ型の関係は固定であり，プログラマがこの関係を変更することはできない．eJSTK でカスタマイズを行う場合は，各内部データ型に関する具体的な表現を提供する必要がある．以下では，内部データ型の具体的な表現を**データ型表現**という．eJSVM のデフォルト設定は，データ型表現を表 3.3 のように定義している．デフォルト設定では，各内部データ型は一つのデータ型表現を持つが，内部データ型とデータ型表現の関係は一対一の関係に制限されない．プログラマは，一つの内部データ型に対して複数のデータ型表現を定義することができる．さらに，プログラマは VM 内部で使用する新しいデータ表現を定義することができる．例えば，6 文字以下の短い文字列を 1 ワードに埋め込むデータ表現を作ることができる．

eJSVM は，値の内部データ型を判別するために，ポインタタグとヘッダタグを使う．値



(a) タグ付きポインタ

(b) データ構造中のヘッダタグ

図 3.1　eJSVM 内部のポインタタグとヘッダタグ

を保持するデータ構造は，ヒープ領域内に，CPU のアーキテクチャによって決められたアラインメントの制約に従って保持される．64 ビットアーキテクチャの場合，全てのデータは 8 バイト区切りで整列されるため，データを指すポインタの下位 3 ビットは，必ずゼロになる．そのためこれらのビットには，データ型を表現するためのポインタタグを保持することができる．このポインタタグでは，最大で 8 つの異なる値を表現することができる．これとは別に，eJSVM はヒープ中のデータ構造に含まれるヘッダタグも使用する．現状の eJSVM は 64 ビットアーキテクチャのみに対応しているため，本論文ではポインタタグのビット数は 3 ビットであることを想定する．ただし，eJSTK の設計はポインタタグのビット数に依存していない．

図 3.1 に，eJSTK におけるタグ付きポインタとヘッダタグの概観を示す．eJSVM の内部では，JavaScript の値は基本的にタグ付きポインタが指すデータである．ただし，一部の内部データ型に関しては，タグ付きポインタ自体に即値としてデータを埋め込む．即値とし

て埋め込まれるのは，整数のような，ヒープ内のデータで表現する必要のないデータ型である．タグ付きポインタがポインタである場合は，ポインタが指すデータ構造のヘッダにヘッダタグが含まれている．表 3.3 に，デフォルト設定の eJSVM における各データ型表現に関して，ポインタタグとヘッダタグの値を示している．

もし，あるデータ型表現が，表 3.3 の `normal_string` のようにポインタタグを他のデータ型表現と共有していないのであれば，そのデータ型表現であるか否かは，ポインタタグをチェックするだけでわかる．以下では，そのようなポインタタグを**ユニークな**ポインタタグという．それに対して，ポインタタグがユニークではなく，他のデータ型表現と共有している場合は，ポインタタグとヘッダタグの両方をチェックする必要がある．これは，ポインタタグがユニークである場合に比べて，ヘッダタグに対して間接的にアクセスする必要があるため，実行にコストがかかる．そのため，VM 内で行われる型ディスパッチを効率的に行うには，頻繁に使用されるデータ型表現に対してユニークなポインタタグを割り当てることが重要である．ただし，タグ付きポインタに即値を埋め込むデータ型には，ユニークなポインタタグを割り当てられていなければならない．

第 4 章

提案システムの全体像

eJSTK は，データ型に注目した VM のカスタマイズを行う方法をプログラマに提供する．eJSVM の行うカスタマイズによって，eJSVM の型ディスパッチ処理は単純になり，VM の実行速度が速くなることが期待できる．さらに，eJSVM のコードのうち使用されないデータ型を処理するコードは生成されないため，カスタマイズされた eJSVM のサイズは小さくなる．これは，組込みシステムの制限された計算資源の点から非常に重要である．

図 4.1 に eJSTK の全体像を示す．プログラマは，内部データ型とデータ型表現の対応と，データ型表現のポインタタグとヘッダタグの値の割り当てを記した，**データ型仕様**を与える必要がある．またプログラマは，全ての命令に関してオペランドに与えることのできるデータ型を定義した，**オペランド仕様**も与える．C 言語で書かれた eJSVM のソースコードは三つのカテゴリに分類することができる．一つ目は，eJSTK が行うカスタマイズとは独立した部分のコード（固定コード）である．これはフレームワークが提供するため，プログラマが詳細を知る必要はない．二つ目は，与えられたデータ型仕様を基に eJSTK が生成したコード（データ型定義コード）であり，主な内容は述語マクロの定義である．三つ目は，インタプリタのメインループで実行される VM 命令のコード（命令コード）である．これは，**命令定義**とデータ型仕様，オペランド仕様の三つから eJSTK の命令コード生成器によって生成される．ここで，命令定義は各 VM 命令について，オペランドのデータ型に注目して振る舞いを記述したもので，本フレームワークが提供する．

eJSVM は内部データ型を，主にインタプリタと型変換関数，ガベージコレクタのトレーサ，組込み関数の四つの場所で使用する．現状の eJSTK は，効率的な型ディスパッチャを生成するのはインタプリタに含まれるものに限定する．これ以外の部分に含まれる型ディス

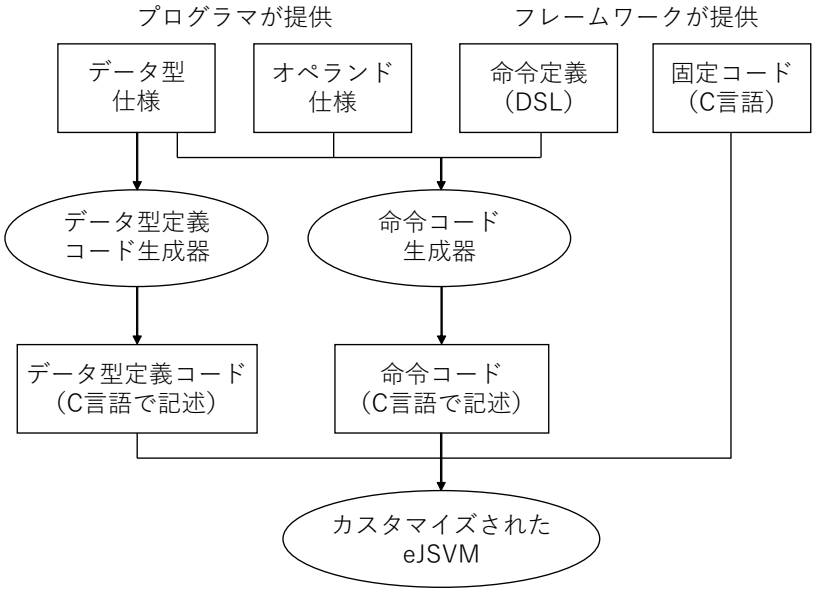


図 4.1　eJSTK の全体像

パッチャについても効率の良いコードを生成することは今後の課題である．

4.1　データ型仕様

eJSVM のデフォルト設定におけるデータ型仕様の一部を図 4.2 に示す．各行はコロン「:」で二つの部分に分けられている．コロンの左側は，内部データ型かデータ型表現の名前である．内部データ型の名前（`string` など）とデフォルトのデータ型表現の名前（`normal_string` など）は予約語である．

コロンの左側が内部データ型の名前であったとき，右側にはその内部データ型に対応するデータ型表現の名前を列挙する．このとき，各データ型表現の名前の先頭には，プラス記号「+」をつける．図 4.2 の 1 行目から 9 行目の記述は，各内部データ型はデフォルトのデータ型表現のみを使用することを意味する．コロンの左側がデータ型表現であったときは，その右側にはデータ型表現に割り当てるポインタタグとヘッダタグの値を次の書式で指定する．

ptag_name(*ptag_val*)/*htag_name*(*htag_val*)

ここで，*ptag_name* は，ポインタタグの値につける名前であり，*ptag_val* はそのポインタタグを二進数で表現した値である．*htag_name* は，ヘッダタグの値につける名前で，*htag_val*

```
1  string:                +normal_string
2  flonum:                +normal_flonum
3  special:               +normal_special
4  fixnum:                +normal_fixnum
5  simple_object:         +normal_simple_object
6  array:                 +normal_array
7  function:              +normal_function
8  builtin:               +normal_builtin
9  regexp:                +normal_regexp
10 normal_string:         T_STRING(100)/HTAG_STRING(4)
11 normal_flonum:         T_FLONUM(101)/HTAG_FLONUM(5)
12 normal_special:        T_SPECIAL(110)
13 normal_fixnum:         T_FIXNUM(111)
14 normal_simple_object:  T_GENERIC(000)/HTAG_SIMPLE_OBJECT(6)
15 normal_array:          T_GENERIC(000)/HTAG_ARRAY(7)
16 normal_function:       T_GENERIC(000)/HTAG_FUNCTION(8)
17 normal_builtin:        T_GENERIC(000)/HTAG_BUILTIN(9)
18 normal_regexp:         T_GENERIC(000)/HTAG_REGEXP(11)
```

図 4.2　デフォルト設定のときのデータ型仕様の記述の一部

はそのヘッダタグを十進数で表現した値である．*ptag_name* と *htag_name* は，データ型定義コード生成器がマクロ定義を生成するときに使用される.「/」とその左側は，タグ付きポインタが即値ではなくポインタを保持するときのみ必要である．例えば，図 4.2 の 13 行目は，`normal_fixnum` のポインタタグの値は 111 であり，さらにそのデータはタグ付きポインタに即値として保持することを示す．15 行目では，`normal_array` のポインタタグは 000 であり，タグ付きポインタはヘッダタグの値が 7 であるデータ構造を指すことを示している．

```
1 add (-,_,_) accept   // instruction for '+'
2 eq (-,_,_) accept    // instruction for '=='
3 call (_,-) accept    // instruction for function call
```

図 4.3 デフォルト設定のときのオペランド仕様の記述の一部

4.2 オペランド仕様

個々のアプリケーションでは，その実行中に各 VM 命令に対して与えられるデータ型が一部に限られていることが多いと考えられる．例えば，あるアプリケーションは，実行される add 命令のオペランドに対して数値のみが与えられることが考えられる．そのときは，数値以外が与えられた時に実行するコードは VM から省くことができる．このようにして，対象のアプリケーションに対して eJSVM をカスタマイズするには，プログラマは各 VM 命令のオペランドに与えられ得るデータ型を指定した**オペランド仕様**を eJSTK に与える．オペランド仕様の各行は次の書式で記述する．

instruction_name(*operand_type*, ...) *action*

ここで，*instruction_name* は VM 命令の名前である．“(*operand_type*, ...)”は入力のオペランドのデータ型の組み合わせを指定する．*operand_type* は，内部データ型の名前かデータ型表現の名前，もしくは「_」か「-」を記述する.「_」はすべてのデータ型を,「-」は入力のオペランドではないことを意味する．指定されたデータ型の値が VM 命令のオペランドに与えられたときの振る舞いは *action* で指定する．*action* に指定できるのは，accept と error，unspecified のどれかである．指定されたデータ型が与えられたときに普通の処理を実行する場合は accept を指定する．error か unspecified を指定した場合は，それに対応するコードは VM から省かれる．error と unspecified の違いは，指定されたデータ型が与えられたときに VM がエラーを出力するかそうでないかである．

デフォルトの設定では，すべての命令はオペランドにどのデータ型をも受理する．図 4.3 に，デフォルト設定で使用する，どのデータ型も制限しないオペランド仕様の一部を示す．

⟨instruction-definition⟩ ::=
  `\inst` ⟨instruction-name⟩ `(` ⟨operand-list⟩ `)` ⟨body⟩
⟨instruction-name⟩ ::= ⟨string⟩
⟨operand-list⟩ ::= ⟨operand⟩ (`,` ⟨operand⟩)∗
⟨operand⟩ ::= ⟨operand-type⟩ ⟨operand-variable⟩
⟨body⟩ ::= ⟨clause⟩∗
⟨clause⟩ ::=
  ⟨when-clause⟩ | ⟨otherwise-clause⟩ | ⟨prologue⟩ | ⟨epilogue⟩
⟨when-clause⟩ ::= `\when` ⟨condition⟩ `\{` ⟨c-program⟩ `\}`
⟨otherwise-clause⟩ ::= `\otherwise` `\{` ⟨c-program⟩ `\}`
⟨prologue⟩ ::= `\prologue` `\{` ⟨c-program⟩ `\}`
⟨epilogue⟩ ::= `\epilogue` `\{` ⟨c-program⟩ `\}`
⟨condition⟩ ::=
  ⟨atomic-condition⟩ | ⟨compound-condition⟩ | `(` ⟨condition⟩ `)`
⟨atomic-condition⟩ ::= ⟨operand-variable⟩ `:` ⟨VM-datatype⟩
⟨compound-condition⟩ ::=
  ⟨condition⟩ `&&` ⟨condition⟩ | ⟨condition⟩ `||` ⟨condition⟩
⟨operand-type⟩ ::=
  `Register` | `Value` | `Subscript` | `Immediate` | `Displacement`
⟨operand-variable⟩ ::= variable name
⟨VM-datatype⟩ ::= name of VM-datatype
⟨c-program⟩ ::= fragment of C codes

図 4.4 命令定義用 DSL の文法

4.3 命令定義

eJSTK には，各 VM 命令について DSL で記述された eJSVM の命令定義が含まれている．この命令定義は命令コード生成器によって処理され，VM 命令のコードになる．

表 4.1 命令定義中の仮引数の型

型の名前	説明
`Register`	レジスタの場所
`Value`	値（タグ付きポインタ）
`Subscript`	添字を表現する整数
`Immediate`	`fixnum` 型と `special` 型の即値
`Displacement`	ジャンプ先の命令を指す値（ジャンプ命令などに使われる）

4.3.1 命令定義用 DSL

図 4.4 に DSL の文法を示す．一つの命令の定義は，⟨instruction-definition⟩ で定義する．⟨instruction-name⟩ は定義する VM 命令の名前，⟨operand-list⟩ はその VM 命令のオペランドを記述し，⟨body⟩ で処理を定義する．

本 DSL には二つの特徴がある．一つは，命令定義の記述の中では，与えられたすべてのオペランドに対して，仮引数を使って簡単にアクセスできることである．仮引数は，その用途に合わせて “型” を持つ．表 4.1 に，仮引数の型を示す．⟨operand-list⟩ で，各仮引数に関していずれかの型を指定する．

もう一つの特徴は，命令の振る舞いを，オペランドに与えられる内部データ型の組み合わせごとに実行する C 言語のコード断片で記述できることである．これは，⟨when-clause⟩ と ⟨otherwise-clause⟩ で行う．⟨when-clause⟩ は，与えられたオペランドのデータ型が ⟨condition⟩ で指定された条件を満たすとき，⟨c-program⟩ で記述された C 言語のコードが実行されるという意味である．⟨otherwise-clause⟩ は，どの ⟨when-clause⟩ の条件も満たさない場合に実行する処理を定義するときに使用する．オペランドのデータ型の条件 ⟨condition⟩ は，データ型表現の名前を使うのではなく，内部データ型の名前を使うことで，データ型の表現に依存せずに記述することができる．

本 DSL は利便性のために，⟨prologue⟩ と ⟨epilogue⟩ を用意している．これは，生成した

コードの前と後に挿入されるコード断片を指定するものである．典型的には，VM 命令中で使用するマクロを ⟨prologue⟩ の中で定義し，⟨epilogue⟩ の中ではそれを未定義に戻すために用いる．

4.3.2 add 命令の例

例として，図 4.5 に DSL で記述された add 命令の定義の一部を示す．add 命令の三つのオペランドは，実際にはレジスタ番号が与えられるが，命令定義では変数になる．add 命令の命令定義では，第一オペランドは結果を格納する変数 dst である．加算演算子の左オペランドと右オペランドの値に対応する第二，第三オペランドは変数 v1 と変数 v2 である．図 4.5 の定義では，五つに分岐する型ディスパッチが定義されている [*1]．

図 4.5 の 2 行目から 5 行目は，v1 と v2 の両方が fixnum 型であるときの処理を定義しており，実際にその内部データ型の組み合わせが与えられたときは，3 行目と 4 行目の C 言語のコード断片が実行される．

15 行目から 19 行目は，v1 が Object 型かつ v2 がプリミティブ型であるときに実行する処理を定義している．JavaScript の加算演算子は，オペランドに Object 型のデータが与えられた場合は，そのデータの to_string メソッドを実行してプリミティブ型に変換し，それに対してもう一度加算演算子を適用するという仕様になっている．そのため，17 行目で v1 を型変換した後，18 行目で命令の先頭にジャンプしている．ここで，add_HEAD は add 命令の定義から生成される VM 命令のコードに付くラベルである．eJSTK は，各 VM 命令ごとに *insn*_HEAD （*insn* は命令の名前）というラベルを定義する．これを使えば再度型ディスパッチするような場合に，同じコード断片が複数の箇所で現れないようにでき，インタプリタのサイズが大きくなるのを抑えることができる．

[*1] eJSVM の add 命令の定義は 8 通りの場合に分岐するが，ここでは説明を簡単にするために 5 通りにしている．

```
1  \inst add (Register dst, Value v1, Value v2) // v1とv2はレジスタ内の値
2  \when v1:fixnum && v2:fixnum \{
3    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
4    dst = cint_to_number(s);
5  \}
6  \when v1:string && (v2:fixnum || v2:flonum || v2:special) \{
7    v2 = to_string(context, v2);
8    goto add_STRSTR;
9  \}
10 \when v1:string && v2:string \{
11 add_STRSTR:
12   dst = cstr_to_string2(context, string_to_cstr(v1),
13         string_to_cstr(v2));
14 \}
15 \when (v1:simple_object || v1:array || ...) &&
16       (v2:fixnum || v2:flonum || v2:special) \{
17   v1 = object_to_string(context, v1);
18   goto add_HEAD;
19 \}
20 \otherwise \{
21   double x1 = to_double(context, v1);
22   double x2 = to_double(context, v2);
23   dst = double_to_number(x1 + x2);
24 \}
25 ...
```

図 4.5 DSL で書かれた add 命令の定義

4.4 カスタマイズの例

この節では，二つの例を使って，eJSTK でどのようにカスタマイズを行うかを説明する．

```
15 | normal_array:   T_ARRAY(010)/HTAG_ARRAY(7)
```

図 4.6 配列型に特化したデータ型仕様の記述の一部

```
1 | add (-,fixnum,fixnum) accept
2 | add (-,flonum,fixnum) accept
3 | add (-,flonum,flonum) accept
4 | add (-,string,string) accept
5 | add (-,_,_) error
```

図 4.7 例 2 のオペランド仕様の記述の一部

4.4.1 例 1: 配列型に対するユニークなポインタタグの割り当て

配列を頻繁に使用するアプリケーションを実行することを想定する．この例では，eJSVM を配列型を特別に扱うカスタマイズを行う．デフォルト設定において，データ型表現 `normal_array` のポインタタグの値は 000 であり，これは JavaScript の各種オブジェクトで共有している．この例では，`normal_array` に対してユニークなポインタタグ 010 を与えることによって，`normal_array` に高速にアクセスできるようにする．

図 4.6 に，このカスタマイズを行うときのデータ型仕様を示す．デフォルト設定との違いは 15 行目の記述のみであり，`normal_array` に対してユニークなポインタタグ 010 を割り当てている．オペランド仕様はデフォルトの設定と同じである．

4.4.2 例 2: オペランドに与えられるデータ型の制限

ここでは，四則演算子と比較演算子について，オペランドに与えるデータ型が限られているアプリケーションを実行することを想定する．例えば，このアプリケーションにおいて，`add` 命令では両方のオペランドに `fixnum` 型か `flonum` 型のどちらか，あるいは両方のオペランドに `string` 型かしか与えられないと想定する．

図 4.7 に，このアプリケーションに特化したオペランド仕様を示す．このオペランド仕様

の add 命令は，上で述べたデータ型の組を処理できるが，その他のデータ型の組が与えられたときはエラーを起こすことが指定されている．図 4.7 のオペランド仕様を使う場合であっても，データ型仕様は任意のものを使用できる．

第 5 章

コードの自動生成

5.1 データ型定義コードの生成

C 言語で書かれた eJSVM のソースコードは，データ型表現を判定する述語マクロを必要とする．述語マクロは `is_`*rep*`(v)`（*rep* はデータ型表現の名前）という形で定義され，タグ付きポインタ `v` がそのデータ型表現かそうでないかを示す値を返す．データ型の表現方法は入力によって異なるため，データ型表現を判定する述語マクロは eJSTK が自動で定義する．同様に，eJSTK はタグ付きポインタが特定の内部データ型であるかそうでないかを調べるための述語マクロも定義する．例えば，`is_object(v)` は，`v` が表 3.1 における Object に対応する内部データ型であるかそうでないかを返す．

図 5.1 に，表 3.1 で示すデフォルト設定が与えられたときに eJSTK が生成する述語マクロの定義の例を示す．ここで，`ptag(v)` は与えられたタグ付きポインタ `v` のポインタタグ，同様に `htag(v)` はヘッダタグを返す．述語マクロを生成する処理は単純である．具体的には，データ型仕様からマクロ定義に必要な情報（タグの名前や値）を得て，各データ型表現について割り当てられたポインタタグがユニークであるかを判定し，必要なマクロ定義を生成する．あるデータ型表現に割り当てられたポインタタグがユニークであった場合，値がそのデータ型表現かどうかを調べるための述語マクロは，ポインタタグのみを利用して判定する．例えば，図 4.2 で示すデータ型仕様では，文字列型のデータ型表現 `normal_string` はユニークなポインタタグ `T_STRING` が与えられているため，生成される述語マクロ `is_string(v)` は，“`(ptag(v) == T_STRING)`”と定義される．逆に，配列型のデータ型表現 `normal_array` に割り当てられているポインタタグ `T_GENERIC` は，

```
1  #define T_STRING 4 // stringのポインタタグ
2  #define T_GENERIC 0 // simple_objectとarrayのポインタタグ
3  #define HTAG_SIMPLE_OBJECT 6 // simple_objectのヘッダタグ
4  #define HTAG_ARRAY 7 // arrayのヘッダタグ
5
6  #define is_string(v) (ptag(v) == T_STRING)
7  #define is_simple_object(v) \
8    ((ptag(v) == T_GENERIC) && \
9     (htag(v) == HTAG_SIMPLE_OBJECT))
10 #define is_array(v) \
11   ((ptag(v) == T_GENERIC) && (htag(v) == HTAG_ARRAY))
12
13 #define is_object(v) (ptag(v) == T_GENERIC)
14 #define is_number(v) \
15   ((ptag(v) == T_FIXNUM) || (ptag(v) == T_FLONUM))
```

図 5.1 デフォルト設定のときの述語マクロ

normal_simple_object などのデータ型表現と共有しているため，生成される述語マクロ is_array(v) は，“((ptag(v) == T_GENERIC) && (htag(v) == HTAG_ARRAY))”と定義される．

5.2 命令コード生成器

VM 命令は，変数に格納されたオペランドのデータ型に基づいて，ディスパッチを行う必要のある命令がある．命令コード生成器は，入力されたオペランド仕様に基づき，不要なオペランドのデータ型に対応するコードを除去した型ディスパッチコードを生成する．命令定義中の条件の記述は内部データ型の名前を使って書かれているため，命令コード生成器は内部データ型に対応するデータ型表現に置き換えて解釈し，データ型表現で判別するような

コードを生成する．

命令コード生成器は，生成する C 言語のコードの効率を高め，サイズを小さくするためにユニークなポインタタグが割り当てられているデータ型のヘッダタグにはアクセスすることがないように型ディスパッチのコードを生成する．また，命令コード生成器は，複数の箇所で同じ C 言語のコード断片が必要な場合，コードサイズを小さくするために，`goto` 文を利用することで同じコード断片が複数個所で存在することを避ける．データ型仕様によってユニークなポインタタグが割り当てられたデータ型表現は，ほぼ同じ頻繁でアクセスされることを仮定する．この仮定は常に成り立つわけではない．この改善は今後の課題である．

生成されるコードはオペランドの数により次のようになる．一つのオペランドの値で型ディスパッチが行われるときは，生成する C 言語のコードは単純である．生成するコードは，まずオペランド値のポインタタグに基づき，`switch` 文を使ってディスパッチを行う．このディスパッチの後，ポインタタグが複数のデータ型表現で共有されている値なのであれば，ヘッダタグの値に基づいて `switch` 文でディスパッチを行う．結果的に生成されたコードは，最大で二度の `switch` 文によるディスパッチを行う．

二つのオペランドの値で型ディスパッチを行う場合，命令コードを生成する処理は複雑になる．命令コード生成器は型ディスパッチコードを生成するために，決定木を構築し，その上で最適化を行う．ここで行う最適化は，[6] で述べられている方法を用いる．オペランド仕様から，その命令に与えられるオペランドのデータ型表現の組み合わせを全て求める．次に，これらの組み合わせをパスとして持つような，ポインタタグやヘッダタグの検査を内部節点とし，C 言語のコードを葉とする決定偽を作る．この決定偽をボトムアップに調べ，まとめても意味が変わらない部分木同士をまとめる．最後に決定グラフ中の出力辺が一つしかない節点を削る．

第 6 章

オペランド仕様の生成

6.1 eJSTK の問題点

アプリケーションの実行において VM 命令のオペランドに実際には与えられる内部データ型を，オペランド仕様で error と指定してしまうとアプリケーションは動作しない．逆に，VM 命令のオペランドに与えられないデータ型を accept と指定すると，それを処理するコードが VM に残り，VM のサイズが必要以上に大きくなってしまう．

しかし，適切なオペランド仕様を定義することは難しい．特定のアプリケーションに合わせてできるだけ VM が小さくなるようなオペランド仕様を作るためには，eJSTK が生成する VM が扱う内部データ型や，生成される VM 命令のコードに関する理解が必要だからである．特に，型変換後に再度型ディスパッチを行うことがある VM 命令が対応するべきオペランドのデータ型を特定することは難しい．例えば，図 4.5 のような定義の add 命令の第二オペランドに配列，第三オペランドに整数が与えられる場合，15 行目から 19 行目のように，型変換を行った後に命令の先頭にジャンプし，再度型ディスパッチを行う．このような場合，add 命令が受理しなければいけないオペランドの内部データ型は，一度目の型ディスパッチの時点での内部データ型（array 型と fixnum 型の組）と，二度目の型ディスパッチの時点での内部データ型である．多くの場合，array 型を object_to_string 関数で型変換した結果は string 型になるが，型変換時に暗黙的に呼び出される to_string メソッドはオーバーライドすることができるため，どのデータ型になるかわからない．つまり，eJSTK のユーザは実行時に array 型をプリミティブ型に型変換を行った結果，どの内部データ型になり得るかを考慮してオペランド仕様を作る必要がある．このような，複数回の型ディスパッ

チを行う場合を考慮して過不足なくオペランド仕様を定義することは難しく，アプリケーション開発者の負担は大きい．そのため，アプリケーションを実行するために，各 VM 命令がサポートするべきデータ型を自動的に特定できるようにすることが必要である．

6.2　ユニットテストの利用

本研究では，プログラマが作成したユニットテストの実行結果から，各演算子のオペランドに与えられる可能性のあるデータ型（以下，VM 命令の機能という）を特定することを目的とし，これを行うためのプロファイラを開発する．

普通，ユニットテストは，コードを網羅的に検査するように作られる．そのため，運用において実際にアプリケーションを実行したときに使用される内部データ型は，テストの実行でアプリケーションが使用した内部データ型は一致すると期待できる．そのため，本研究ではユニットテストを利用する．

本プロファイラは，JavaScript アプリケーションを対象としたユニットテストフレームワーク Jasmine[*1] を使って作られたユニットテストを利用する．

6.3　Jasmine

多くのアプリケーションの開発で，アプリケーションが正しく動作するかを検査するためにユニットテストが行われている．ユニットテストは，プログラムの機能ごと，多くの場合は関数ごとにテスト実行し，そのときのプログラムの振る舞いが意図したものかを検査する．

ユニットテストでは，ある関数 a と，これを呼び出す関数 b があるとき，この二つの関数を分けてテストを行うことがある．そのような場合，関数 b のテストは，関数 a をテスト用の別の関数に差し替えて行う．このようなテスト用に差し替える仮の関数をモックという．Jasmine は，モックを簡単に作成できるなどの，ユニットテストの作成を支援する機能を備

[*1] https://jasmine.github.io/

```
1  function calc_sum(integers) {
2    var sum = 0;
3    for (var i = 0; i < integers.length; i++) {
4      sum = sum + integers[i];
5    }
6    return sum;
7  }
8
9  function slice_and_sum(integers, l, r) {
10   return calc_sum(integers.slice(l, r));
11 }
```

図 6.1 テスト対象のコード

えている.

Jasmine の利用例として，図 6.1 のコードを対象に，Jasmine を利用して記述したテストを図 6.2 に示す．図 6.1 の関数 `calc_sum(integers)`（関数 a に相当）は，引数 `integers` に渡されるのは整数の配列であることを想定しており，これの要素を全て足し合わせた値を返す．関数 `slice_and_sum(integers,l,r)`（関数 b に相当）は，配列オブジェクトのメソッド `slice` を利用して，整数の配列 `integers` の `l` と `r` で選択された範囲（添字が `l` 以上 `r` 未満）を別の配列として取り出し，それを関数 `calc_sum(integers)` に渡した結果を返す.

Jasmine を使ったテストでは，関数 `describe` で複数のテストをまとめ，テストは関数 `it` に第二引数として渡す関数の中で行う．図 6.2 の 3 行目における関数 `it` の呼び出しは，関数 `calc_sum` のテストを行っている．結果が正しいことの判定は，6 行目のように，関数 `expect` と関数 `expect` が返すオブジェクトのメソッドである `toBe` を利用する.

9 行目の関数 `it` の呼び出しは，関数 `slice_and_sum` のテストを行っている．関数 `slice_and_sum` は，関数 `calc_sum` の実装に依存しているため，ここで関数 `calc_sum`

```
1  describe("example", function() {
2
3    it("test_calc_sum", function() {
4      // 関数calc_sumの動作チェック
5      var actual_value = calc_sum([1, 2, 3, 4, 5]);
6      expect(actual_value).toBe(15);
7    });
8
9    it("test_slice_and_sum", function() {
10     // 関数slice_and_sum内で呼び出す関数calc_sumをモックで差し替える
11     spyOn(__global__, "calc_sum").and.callFake(function(integers) {
12       // 任意の処理（integersが[2, 3]であるかのチェックなどができる）
13       return 5;
14     });
15
16     // 仮の関数calc_sumを使って関数slice_and_sumが実行される
17     var actual_value = slice_and_sum([1, 2, 3, 4, 5], 1, 3);
18     expect(actual_value).toBe(5);
19   });
20
21 });
```

図 6.2　Jasmine を利用して書かれた図 6.1 のテストコード

をモックに差し替える．11 行目のような記述で，関数 `calc_sum` をその場で定義した関数に差し替えることができる．

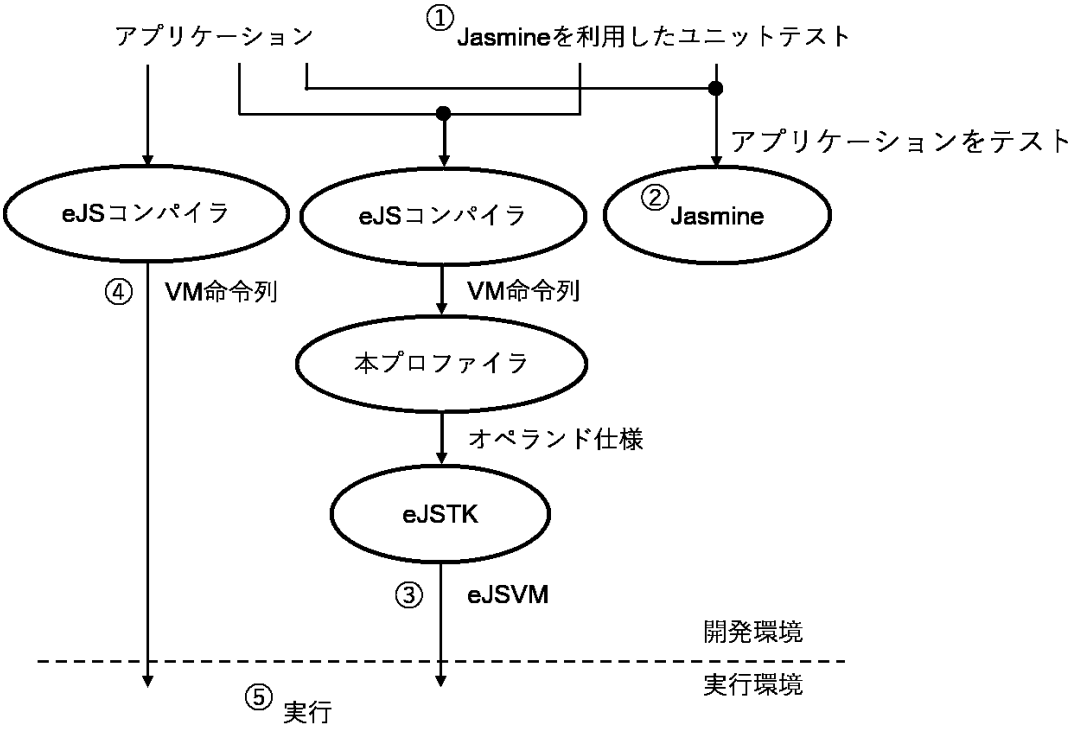


図 6.3　プロファイラを使った開発フロー

6.4　提案するプロファイラの構成

本プロファイラは，実行中に使われた内部データ型を記録する機能を持った VM（以下，Profiling VM）と，Jasmine を利用して作られたユニットテストを実行するための JavaScript プログラム（以下，テストランナー）から構成される．

図 6.3 に，本プロファイラと eJSTK を利用した開発の流れを示す．ユーザは，まず通常の開発通り Jasmine を使ってアプリケーションとユニットテストを実行し，テストを行う（図 6.3②）．本プロファイラは，この図 6.3 の②で使ったユニットテスト（図 6.3①）をそのままを利用する．テストランナーと与えられたアプリケーションとユニットテストをコンパイルし，その結果の VM 命令列を Profiling VM 上で実行する．テストランナーは，`describe` や `it` など，Jasmine と同じ名前の関数を提供しており，Profiling VM はこれを使ってユニットテストを実際に実行する．

Profiling VM は，アプリケーションの実行を監視し，VM 命令の各オペランドに実際に

渡された内部データ型を記録する．例えば，Profiling VM で図 6.2 のユニットテストを実行した場合，関数 calc_sum 内の加算演算子からコンパイルされた add 命令を処理するときに，オペランドに与えられた内部データ型や，それが型変換された後再度ディスパッチが行われた場合はその内部データ型をログに記録する．

オペランドに与えられた内部データ型を記録するのは，加算演算子などの算術演算子からコンパイルされた VM 命令だけではなく，型ディスパッチを行うすべての命令が内部データ型を記録する．関数呼び出しを行う call 命令は，そのような命令の一つである．call 命令は，呼び出す関数をオペランドに受け取り，その内部データ型が function 型か builtin 型かによって実行する処理を変える．図 6.1 の例では，関数 slice_and_sum は，call 命令によって関数 calc_sum を呼び出す．プロファイリング実行時に call 命令が実行されたときは，この呼び出す対象の内部データ型が記録される．

型ディスパッチを行わない VM 命令に関しては，オペランドの内部データ型を記録することはないが，その VM 命令が現れたことは記録する．

このようにして得たプロファイリングの結果からオペランド仕様を作る．このとき，型ディスパッチを行う VM 命令に関してはオペランドに与えられた内部データ型のみを受理するようにする．また，型ディスパッチを行わない命令に関しては，ユニットテストの実行で現れなかった命令は VM に含まれないようにする．このオペランド仕様を eJSTK の入力として与えることで，ユーザ自身でオペランド仕様を記述することなくアプリケーションに特化した小さな VM を得ることができる（図 6.3③）．

最後にユーザは，生成された VM と，アプリケーションをコンパイルして得た VM 命令列（図 6.3④）を実行環境に移し，実行する（図 6.3⑤）．

6.5　プロファイラの実装

基本的な実装は単純であるが，以下の二つの点に注意して設計する必要がある．

1. プロファイリングの結果にテストのコードでしか使われない内部データ型を含んではな

らない

2. ユニットテストの実行と実際の実行で，オペランドに与えられるデータ型が異なる場合がある

以下では，まず基本的な実装を述べ，その後に上で述べた点に関する対応を述べる．

6.5.1　基本的な実装

まず，アプリケーションとユニットテスト，テストランナーの三つの JavaScript プログラムを一つの VM 命令列にコンパイルする．そして，コンパイルによって得た VM 命令列を Profiling VM で実行する．

テストランナーが提供する Jasmine の関数の基本的な動作は，本来の Jasmine と同じである．例えば，`describe` と `it` は第二引数に与えられた関数を実行する．これによって，本プロファイラは与えられたユニットテストを実行することができる．ただし，本プロファイラの目的はユニットテストではないため，テストランナーが提供する関数の振る舞いは本来の Jasmine のものと一部異なる．例えば，図 6.2 の 6 行目では，`expect` と `toBe` を使い，実行結果がユニットテストの期待する値（正解の値）であるかを調べているが，テストランナーはそれらの関数を何もしない関数として定義する．

Profiling VM は，基本的には VM 命令が実行されたときにオペランドに与えられた内部データ型を記録するだけであるが，第 6.1 節で述べたように，型ディスパッチを複数回行う場合を考慮する必要がある．図 4.5 で定義される `add` 命令において，第二オペランドに `array` 型の値が与えられる場合，二度目の型ディスパッチ時点での `v1` と `v2` の内部データ型も受理するようなオペランド仕様を作る必要がある．

そのため開発したプロファイラは，命令の先頭でオペランドの内部データ型を記録するのではなく，型ディスパッチを行う時点のオペランドの内部データ型を記録する．

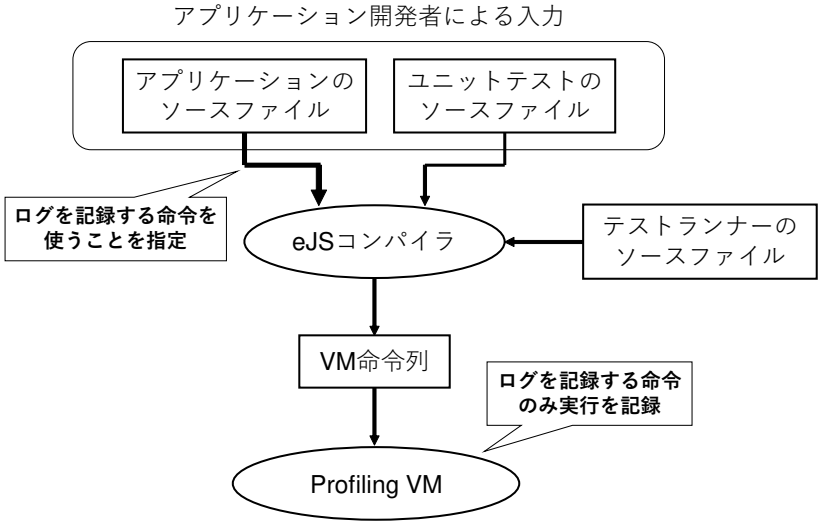


図 6.4　プロファイリング用のコンパイルの様子

6.5.2　アプリケーションのコードとテストのコードの区別

Profiling VM は，アプリケーションとユニットテスト，テストランナーのコードの三つを結合して実行することを想定している．もし，この実行全体をプロファイリングした場合，ユニットテストやテストランナーの中でしか使われない内部データ型を必要と判定してしまう．例えば，関数 `calc_sum` をテストするユニットテストの記述の中に，`string` 型と `string` 型の組み合わせに対して加算演算子を適用しているとする．この場合，`add` 命令に関して関数 `calc_sum` の実際の実行では行われないはずの `string` 型と `string` 型の組み合わせが行われたことになってしまう．

そこで，アプリケーションで使われた内部データ型だけを記録するために，各 VM 命令について，通常の動作をする命令と，第 6.5.1 節で説明した，オペランドに与えられた内部データ型をログに記録した上で通常の命令の実行処理を行う命令の二つを用意する．図 6.4 に示すように，アプリケーションはオペランドに与えられた内部データ型をログに記録する VM 命令を使った命令列にコンパイルする．一方，ユニットテストとテストランナーは通常の VM 命令を使った命令列にコンパイルするようにする．

普通，ユニットテストは，アプリケーションとは別のファイルに記述される．そのため，

記述されたファイルでアプリケーションかユニットテストかを区別するようにする．具体的には，eJS コンパイラに複数のソースファイルを入力するとき，一部のソースファイルに対してログを残す命令を使ってコンパイルすることを指定できるようにする．eJS コンパイラは，複数の JavaScript ファイルを一つの VM 命令列にコンパイルするが，その過程で，指定されたファイルに記述された JavaScript プログラムのコンパイル結果の VM 命令列には，ログを残す VM 命令を使う．これによって，プロファイリング実行時にはアプリケーションが使用する内部データ型のみ記録できる．

6.5.3　プロファイル時と実際の実行の差異とその対応

eJSTK が生成する VM において，ユーザ定義関数は `function` 型，組込み関数は `builtin` 型で表現される．組込みシステムの開発においては，実機上でのみ使用できる組込み関数を使うことがあり，それを使ったアプリケーションをユニットテストするときは，組込み関数をエミュレートするモックを作ることが考えられる．しかし，ユニットテストで組込み関数をモックで差し替えていた場合，モックはユーザ定義関数であるため，プロファイリング実行時と実際の実行で内部データ型が異なることになる．

例として，配列のメソッド `slice` を使う関数 `slice_and_sum` を考える．eJSVM では，`slice` は組込み関数として実装されているため，内部データ型は `builtin` 型である．もし，関数 `slice_and_sum` のユニットテストで，何らかの理由で `slice` をモックで差し替えると，ユーザが定義したモックは `function` 型になる．プロファイリング実行時にこの値が VM 命令のオペランドに与えられたときは `function` 型が与えられたと記録されるが，実際の運用においては `builtin` 型が与えられ，エラーを引き起こす原因になる．

そのため，ユニットテストの実行でオペランドに `function` 型が与えられた命令は，`builtin` 型も受け取るようなオペランド仕様を生成する．これによって VM のサイズは大きくなってしまう可能性があるが，次の理由から実用的にはそのようなことは少なく，大きくなったとしてもほとんど差はない．ほとんどの VM 命令は，オペランドに `function` 型を受け取ったときと `builtin` 型を受け取ったときに実行されるコードを共有していて，これら

の命令では function 型と builtin 型を区別しなくとも，VM のサイズに影響はない．関数呼び出しを行う call 命令とコンストラクタ呼び出しを行う new 命令だけは，function 型と builtin 型で実行される処理が異なる．しかし，ほとんどのアプリケーションは，ユーザ定義関数と組込み関数の両方を呼び出すことが考えられるため，いずれにしろ call 命令は，function 型と builtin 型をサポートすることが多い．new 命令に関しても同様である．

第 7 章

評価

この章では，eJSTK が生成した VM の性能と，オペランド仕様を生成する方法としてユニットテストの実行をプロファイリングすることの妥当性を評価する．

7.1 eJSTK の性能評価

我々は，次の 2 つの観点から eJSTK を評価した．

- 各種カスタマイズの効果
- 他の JavaScript エンジンとの比較

実験に使用した環境は二つあり，これらを以下では，それぞれ ARM，x86 と呼ぶ．ARM の実行環境は，Raspberry Pi 3 であり，CPU は BCM2837 64 ビット（1.2 GHz）を搭載している．OS は Raspbian GNU/Linux 8(Linux kernel4.9.35)，使用したコンパイラは GCC 7.3.0 である．x86 は，Intel(R) Core(TM) i7-6700K CPU（4.00 GHz）を搭載するデスクトップコンピュータである．OS は Ubuntu 14.04.5 (Linux kernel 3.13.0)，使用したコンパイラは GCC 7.3.0 である．どちらの環境においても，VM をコンパイルするときはコードサイズを抑制するコンパイルオプション`-Os` を使用した．x86 は組込みシステムではないが，eJSTK の性能を示すために実験に使用した．

なお，ガベージコレクタが VM の性能に与える影響を抑えるために，全ての実行においてヒープサイズを 10MB に設定した．現状の eJSVM のガベージコレクタは，特にチューニングされていない単純なマークスイープ GC を使っている．更に eJSVM のオブジェクト

モデルは，組込みシステムに特化しておらず，大きなヒープ領域を必要とする．オブジェクトモデルを効率化することは，本研究では扱わず，今後の課題とする．

ベンチマークプログラムには，SunSpider ベンチマーク [33] のバージョン 0.9 に対して，eJSVM 上で実行できるように小さな改変を加えたものを使用した．使用したベンチマークプログラムは，図 7.1 に示すグラフの横軸に示すものである．

ベンチマークプログラムの名前の最初には，そのプログラムの性質を示したカテゴリ名が付けられている．プログラムの性質とそのカテゴリ名は以下のようなものである．

- `access-`が名前に付くベンチマークは，オブジェクトのプロパティか配列に頻繁にアクセスする．
- `bitops-`が名前に付くベンチマークは，整数（eJSVM では `fixnum` 型）に対してビット演算を行う

各ベンチマークに加えた改変は大きく分けて次の二つである．一つは，プログラム中の繰り返し回数を少なくしたことである．もう一つは，プログラム中で使用されるいくつかの組込み関数を使用しないように書き直したことである．例えば，`string-base64` では，テキストをエンコードするために `Math.random` を使用するが，これを手動で作成した擬似乱数生成関数に置き換えた．

我々は，eJSTK を用いて生成した 4 つの VM と，eJSTK を使わずに開発した VM（以下，`handcrafted`）について，VM のバイナリサイズとベンチマークプログラムの実行時間を比較した．四つの VM は，二つのデータ型仕様と二つのオペランド仕様の組み合わせを eJSTK に与えることによって生成した．使用したデータ型仕様は，`default` と `arraytag` である．`default` は，表 4.2 で示したデータ型仕様である．`arraytag` は，第 4.4.1 節で説明した例に対応するデータ型仕様である．`arraytag` では，normal_array にユニークなポインタタグを割り当てている．ここで割り当てたポインタタグの値は，`default` では使用されていない「001」である．

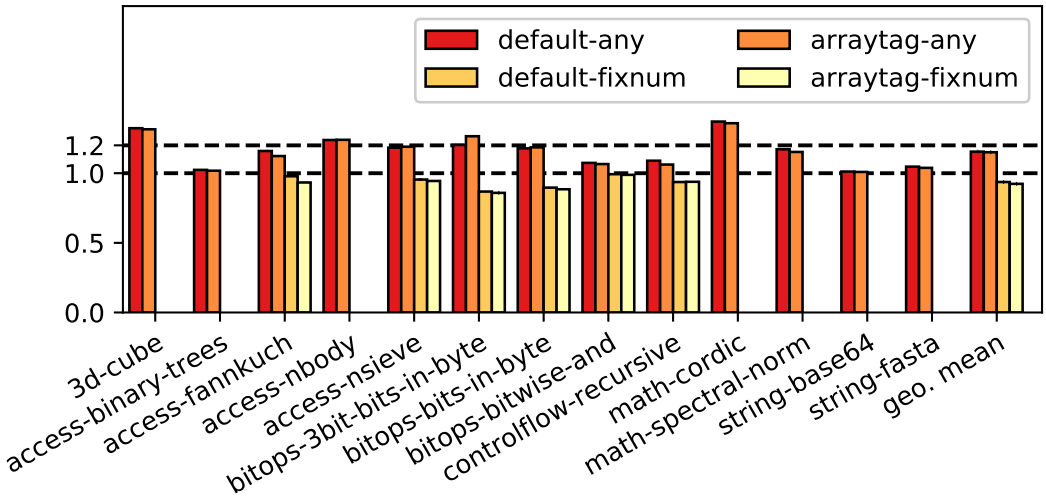
また，使用したオペランド仕様は，any と fixnum である．any で生成した VM は，各 VM 命令のすべてのオペランドに，すべてのデータ型を受理する．fixnum で生成した VM は，四則演算と比較演算に関して，fixnum 型のみ受理する．ただし，add 命令に限り，二つの fixnum 型の組み合わせと，二つの string 型の組み合わせを受理する．もし，サポートしないデータ型がオペランドに与えられた場合，VM はクラッシュする可能性がある．正確には，データ型が与えられたときに実行されるコードが実行される．

以下では生成した四つの VM を，入力のデータ型仕様とオペランド仕様の名前を連結した名前 default-any，arraytag-any，default-fixnum，arraytag-fixnum で表す．

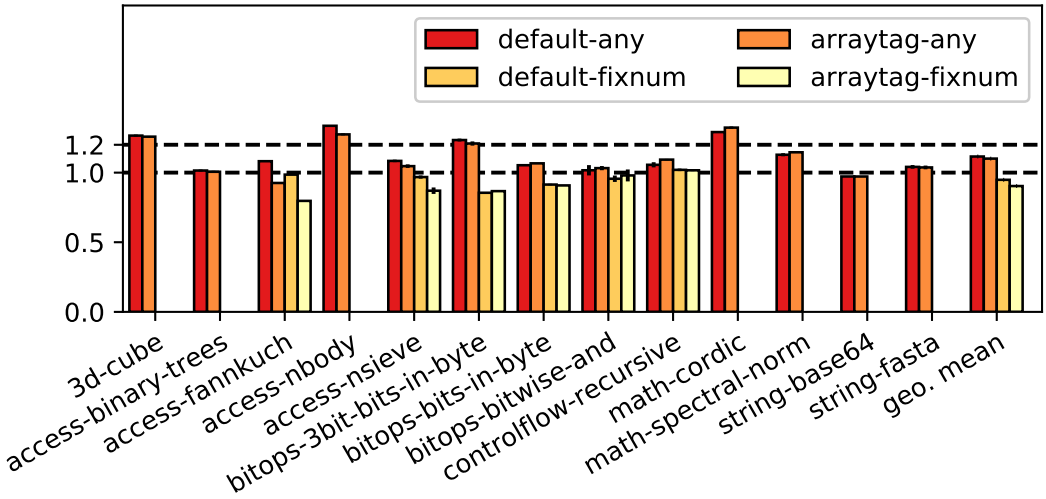
handcrafted VM のデータ型の表現と，各 VM 命令のオペランドが受理するデータ型は，デフォルト設定で生成された VM（default-any）と同じであるが，使用する VM 命令は我々が作成し，手動で可能な限り高速に動作するようにチューニングした．その一つとして，インタプリタループ内の VM 命令は，実行される可能性の高い場合のみを扱い，そうでない場合の処理は別の関数で定義し，それを呼び出すことで実行するようにした．例えば，add 命令では，両オペランドが数値，もしくは文字列でないときの処理は別の関数で実行する．これによって，インタプリタのメインループが小さくなり，実行が高速になる期待できる．

図 7.1 に，ARM 環境と x86 環境の上で，各 VM がベンチマークの実行にかかった時間を示す．ここで示しているすべての値は，handcrafted の実行速度を基準（1.0）として正規化している．

また，図 7.2 には，handcreafted と eJSTK で生成した各 VM のサイズを示す．色付けされている部分は，各 VM のうちインタプリタのサイズである．灰色の部分は，VM 中のインタプリタ以外の部分（組込み関数や GC など）のうち，組込み関数を除いた VM のサイズである．組込み関数を除いたのは，eJSTK は今後，VM に含む組込み関数をアプリケーションの実行に必要なもののみにすることを想定しているためである．



(a) ARM

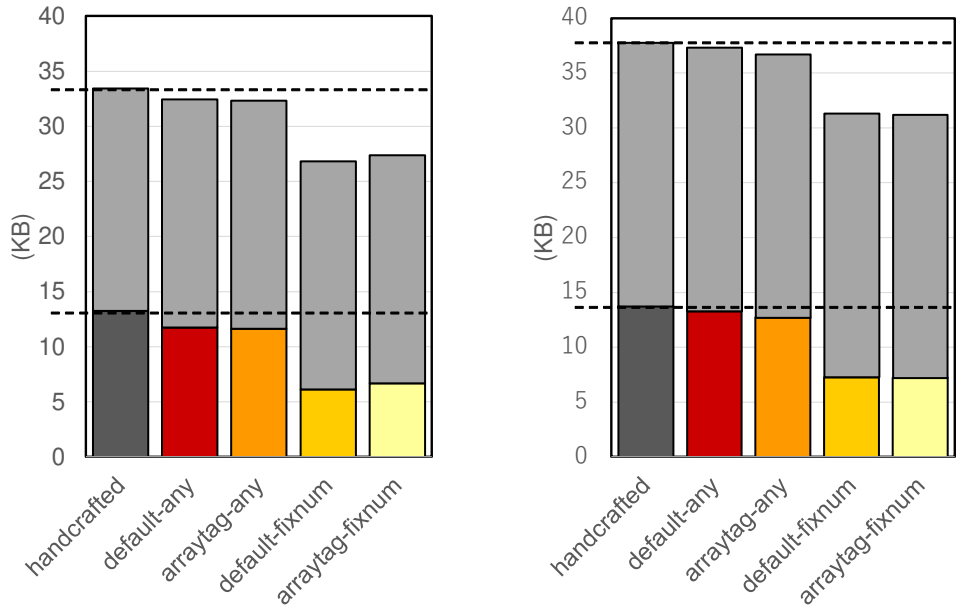


(b) x86

図 7.1　`handcrafted` を基準とした実行時間の比較

7.1.1　VM コード生成によるオーバーヘッド

eJSTK の VM コード生成は，完全に手書きで VM を作成した場合に比べて，オーバーヘッドがあることが考えられる．このオーバーヘッドが実際にどのくらいかを調査するために，`default-any` と `handcrafted` を比較した．VM のサイズは，図 7.2 で示すように，`default-any` と `handcrafted` はほぼ同じであった．ARM 環境での実行速度に関しては，13 のベンチマークプログラムのうち 7 つで `handcrafted` が `default-any` に比べて 1.2 倍



(a) ARM　　(b) x86

図 7.2　生成された 4 つの VM と handcrafted のサイズ.

以上高速であった．このことから，自動生成によるオーバーヘッドは実行速度に関してのみあることがわかった.

7.1.2　データ型の表現をカスタマイズすることによる効果

arraytag-any と arraytag-fixnum は，プロパティアクセス（配列アクセスも含む）を頻繁に行うプログラムに特化した VM である．これら VM は，x86 環境において，配列アクセスを頻繁に行うプログラムの実行が，default-any と default-fixnum のそれぞれと比較して高速であった．arraytag-any と arraytag-fixnum は，normal_array にユニークなポインタタグの値を割り当てているため，各 VM 命令のオペランドに配列が与えられた場合は，ヘッダタグにアクセスせずに型ディスパッチを行うことができる．実際に，access-fannkuch の実行中に起こるヘッダタグへのアクセスの回数は，handcrafted の 2,782,774 回から，arraytag-any では 245 回にまで少なくなっている．しかし，ARM 環境では，ヘッダタグへのアクセスにかかるコストが非常に小さく，実行速度に変化がなかった.

7.1.3　オペランドに与えられるデータ型を制限することの効果

default-fixnum と arraytag-fixnum は，オペランドに与えることのできるデータ型を制限しているため，他の VM に比べてサイズが小さく，実行速度は速い．データ型を制限した default-fixnum は，実行できたすべてのベンチマークにおいて，一切の制限を行っていない default-any よりも高速かつサイズが小さかった．arraytag-fixnum と arraytag-any を比較しても，同様のことがいえる．このことから，オペランドのデータ型を制限することは効果的であるといえる．

オペランドのデータ型を制限した VM は，制限したデータ型を使用するプログラムを実行することはできないが，カスタマイズした VM はその VM 上で実行する特定のプログラムのみに特化しているため，問題にはならない．

7.1.4　他の JavaScript エンジンとの比較

我々は，ARM 環境で，`default-any` と他の組込みシステム用 JavaScript エンジンである JerryScript [34]（バージョン 1.0）と，V7 [35]（バージョン 3.0）を比較した．JerryScript と V7 は，どちらもデフォルトの設定で実行した．JerryScript のデフォルト設定のヒープサイズは 512KB 固定である．V7 の初期のヒープサイズは 22KB であり，実行時に必要に応じて拡張される．

表 7.1 に，各ベンチマークプログラムの実行時間とガベージコレクションにかかった時間を示している．ここで示す全体の実行時間は，ガベージコレクションを実行した時間も含む．JerryScript は参照カウントガベージコレクションを使っており，ガベージコレクションの実行時間を計測できなかったため，ここでは全体の実行時間のみを示している．

JerryScript と比較すると，eJSVM は同程度の速度であった．13 のベンチマークプログラムのうち 8 つは，eJSVM の方が実行が速かった．しかし，`string-base64` に限り eJSVM は非常に遅かった．これは，eJSVM は文字列を効率よく生成できないためである．

V7 は，ガベージコレクション以外の純粋な計算時間を比較しても，eJSVM よりも実行

表 7.1 他の組込みシステム用 JavaScript エンジンとの比較．括弧の外の値は全体の実行時間（秒），括弧の中の値はガベージコレクションの実行時間（秒）．

ベンチマークプログラム	eJSVM	JerryScript	V7
3d-cube	4.81 (0.13)	5.3	91.43 (21.04)
access-binary-trees	3.01 (0.08)	3.3	137.06 (116.36)
access-fannkuch	6.49 (0.00)	13.0	276.16 (40.15)
access-nbody	8.50 (0.13)	6.6	74.46 (15.35)
access-nsieve	1.73 (0.01)	failed	> 3600
bitops-3bit-bits-in-byte	1.85 (0.00)	3.5	65.64 (12.71)
bitops-bits-in-byte	3.21 (0.00)	5.1	91.97 (10.76)
bitops-bitwise-and	9.68 (0.00)	6.2	46.75 (0.00)
controlflow-recursive	1.98 (0.00)	2.3	310.17 (278.44)
math-cordic	5.64 (0.04)	7.4	113.83 (19.28)
math-spectral-norm	3.08 (0.02)	3.3	164.14 (75.63)
string-base64	61.72 (0.99)	8.3	421.41 (48.60)
string-fasta	9.54 (0.19)	7.3	112.88 (29.03)

速度が遅かった．V7 のプロパティは連想リストで保持する実装となっており，プロパティアクセスや配列アクセスを行うときはこのリストを辿らなければいけないことに起因していると考えている．

eJSVM は，メモリ管理システムの影響を小さくするために，大きなヒープを使用した．しかし，もし JerryScript の実行時間の半分がガベージコレクションに費やされているとしても，eJSVM は JerryScript と実行速度は同程度であるといえる．また，eJSVM の全体の実行時間は，V7 の純粋な実行時間と比較しても高速であった．このように，ヒープを大きく設定したことは，結論に大きな影響は与えていない．

まとめると，eJSVM は他の組込みシステム用 JavaScript エンジンと比較しても同程度の性能を持つといえる．

我々は，x86 環境においても同様に他の JavaScript エンジンとの比較を行った．比較対象は，Rhino（バージョン 1.7.9）と SpiderMonkey(バージョン C24.2.0)，V8（バージョン 7.0.276.28-node.5）である．これらのエンジンはすべてデフォルト設定のものを使用した．

表 7.2　デスクトップ PC 上で動作することを想定した他の JavaScript エンジンとの比較．括弧の外の値は全体の実行時間（秒），括弧の中の値はガベージコレクションの実行時間（秒）．

ベンチマークプログラム	eJSVM	Rhino	SpiderMonkey	V8
3d-cube	0.357 (0.027)	0.398 (0.016)	0.037 (0.0)	0.039 (0.003)
access-binary-trees	0.209 (0.006)	0.245 (0.007)	0.012 (0.0)	0.011 (0.002)
access-fannkuch	0.540 (0.000)	0.695 (0.010)	0.033 (0.0)	0.034 (0.001)
access-nbody	0.636 (0.034)	0.360 (0.009)	0.014 (0.0)	0.015 (0.002)
access-nsieve	0.133 (0.000)	1.511 (0.075)	0.015 (0.0)	0.020 (0.002)
bitops-3bit-bits-in-byte	0.121 (0.000)	0.157 (0.008)	0.005 (0.0)	0.006 (0.001)
bitops-bits-in-byte	0.255 (0.000)	0.300 (0.008)	0.021 (0.0)	0.016 (0.001)
bitops-bitwise-and	0.546 (0.000)	0.407 (0.005)	0.024 (0.0)	0.017 (0.001)
controlflow-recursive	0.145 (0.000)	0.158 (0.006)	0.011 (0.0)	0.013 (0.001)
math-cordic	0.399 (0.014)	0.351 (0.011)	0.016 (0.0)	0.024 (0.002)
math-spectral-norm	0.218 (0.008)	0.157 (0.007)	0.007 (0.0)	0.009 (0.002)
string-base64	23.373 (0.100)	0.613 (0.011)	0.021 (0.0)	0.043 (0.007)
string-fasta	1.057 (0.022)	0.625 (0.011)	0.106 (0.0)	0.058 (0.003)

Rhino においては，Java の初期ヒープサイズは 118MB で，実行時に最大 207MB まで拡張した．V8 の初期のヒープサイズは 4.2MB であり，実行中には最大 11.2MB まで拡張した．SpiderMonkey のヒープサイズについては調査することができなかった．表 7.2 に結果を示す．ここで示す実行時間は，ガベージコレクションの実行時間も含んでいる．

eJSVM は，Rhino と同程度の実行速度であった．しかし，SpiderMonkey や V8 と比較すると非常に遅かった．eJSVM は小さなメモリしか持たないようなシステムを対象とするので，デスクトップ PC 上の JavaScript VM が行う最適化は取り入れることができないものもあるが，eJSVM の性能向上は今後の課題である．

7.2　オペランド仕様を生成する手法の妥当性の評価

ユニットテストのプロファイリングから必要な VM 命令の機能を特定する手法の妥当性を検証した．我々は，Raspberry Pi 上で動作する組込みアプリケーションに対して，本手

法と型推定を用いた方法の両方でそれぞれオペランド仕様を生成し，それらをもとに生成した VM のサイズを比較した．

7.2.1　型推定を用いた方法

比較の対象として，VM 命令列に対して型推定を行い，その結果からオペランド仕様を生成するシステムを作成した．このシステムは，各 VM 命令のオペランドに適用され得る内部データ型の集合を特定し，その内部データ型に対応する機能のみをサポートする．しかし，JavaScript における型推定は難しい．本研究で作成した型推定のシステムは，単純なアルゴリズムによるものであり，限られた範囲でしか型を推定できない．

作成した型推定システムは，アプリケーションを VM 命令のレベルで記号的に実行する．このシステムにおける記号は，内部データ型の集合である．型環境は，プログラムのある実行時点でレジスタやローカル変数がどのような内部データ型であり得るかを示す．具体的にはレジスタやローカル変数から内部データ型の集合へのマップである．記号実行器は，VM 命令を実行したときにオペランドに与えられたレジスタにあり得る内部データ型の集合を型環境から調べ，その結果を記録する．また，VM 命令のもたらす副作用に従って型環境を更新する．記号実行が終わると，各 VM 命令に記録した，オペランドに与えられ得る内部データ型の集合から，オペランド仕様を作る．

型推定の方針として，グローバル変数と関数の返り値，プロパティアクセスで得られる値の内部データ型は未知（全ての内部データ型があり得る）とする．また，JavaScript では呼び出された関数の中で呼び出した側のローカル変数を書き換えることができるため，関数呼び出しがあればその時点でローカル変数の内部データ型は未知になるものとする．

7.2.2　評価に使用したアプリケーション

評価用のアプリケーションには，表 7.3 に示す二つを作成した．これら二つのアプリケーションの実行環境として，ARM プロセッサを搭載したシングルボードコンピュータである

表 7.3　作成したアプリケーション

名前	アプリケーションの行数	テストの行数	説明
pimorse	113	78	文字列をモールス信号に変換，LED で出力
hygrothermograph	459	316	温湿度を取得し，LCD ディスプレイで出力

Raspberry Pi，OS は Raspbian 上で動作することを想定している．

一つ目のアプリケーションは，文字列からモールス信号への変換器である（以下，pimorse）．このアプリケーションは，コマンドラインから入力された文字列をモールス信号に変換し，更にその信号パターンを Raspberry Pi の GPIO に接続された LED で再生する．Raspberry Pi 上で動作する C 言語で記述されたモールス信号変換器のソースコード [*1] が公開されており，これを JavaScript に移植した．移植したプログラムの行数は 113 行であった．また，Jasmine を利用したテストを作成し，その行数は 78 行であった．

二つ目のアプリケーションは，温湿度計である（以下，hygrothermograph）．このアプリケーションは，GPIO に接続された温湿度センサ（dht11）から温湿度情報を受信し，その情報を I2C で接続されたキャラクタ LCD で表示する．温湿度センサの制御プログラム [*2] とキャラクタ LCD の制御プログラム [*3] は公開されている C/C++コードを JavaScript に移植した．コードの規模に関しては，移植したプログラムの行数は 459 行，Jasmine を利用したテストの行数は 316 行であった．

7.2.3　結果

表 7.4 に，本手法を適用した場合と型推定を用いる方法を適用した場合について，必要と判定された演算子の機能を示す．スペースの都合上，表 7.4 に示しているのは四則演算と等

[*1] https://github.com/gabolander/pimorse
[*2] http://osoyoo.com/ja/2017/07/06/dht11/
[*3] https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library

表 7.4　各手法で必要と判定された機能

アプリケーション	オペランド仕様	演算子とサポートするオペランドのデータ型											
		+			-		*	/	==		===		
		fixnum,fixnum	string,string	simple_object,simple_object	fixnum,fixnum	string,string	fixnum,fixnum	fixnum,fixnum	fixnum,fixnum	string,string	fixnum,fixnum	string,string
pimorse	手作業で作成	○			○		○					○
	本手法で生成	○			○		○					○
	型推定を用いて生成	○			○	○	○				○	○
	VM 命令単位	○	○	○	○	○	○				○	○
hygrothermograph	手作業で作成	○	○		○			○	○			
	本手法で生成	○	○		○			○	○			
	型推定を用いて生成	○	○	○	○		○	○	○	○		
	VM 命令単位	○	○	○	○	○	○	○	○	○		

価演算を行う演算子の機能のみである．各手法について，○と記されているものが，必要と判定された演算子の機能である．また，図 7.3 に，各手法で作ったオペランド仕様を用いて生成された VM のうち，インタプリタのサイズを示す．ここで示すインタプリタのサイズが，第 7.1 節で使用した eJSVM とは異なるのは，eJSVM のバージョンの違いによるものである．VM はインタプリタの他，GC などのランタイムシステムや組込み関数で構成されるが，インタプリタ以外はオペランド仕様によらずに固定のプログラムが使用されるので比較の対象としない．図 7.3 に示しているフルセットは，一切の演算子の機能を省いていないフルセットの VM を指す．

後の議論のため，表 7.4 と図 7.3 には更に二つの系列を示している．一つは，我々がアプリケーションを読んで手動で作成したオペランド仕様から生成した VM「手作業で作成」である．もう一つは，VM 命令単位で取捨選択した VM「VM 命令単位」である．この VM

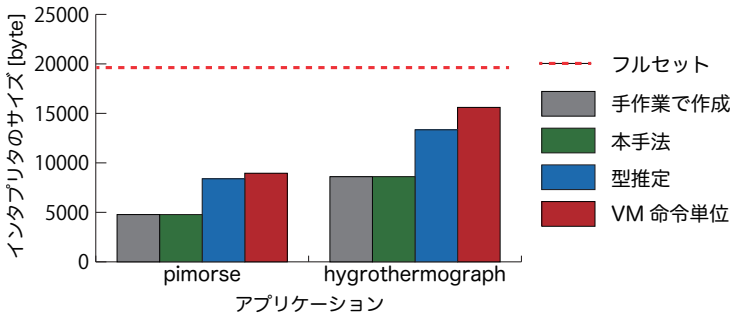


図 7.3　eJSTK が生成した eJSVM 全体のうちインタプリタ部分のサイズ

ではコンパイルされた VM 命令列の中に現れる全ての命令について，あらゆる内部データ型をサポートする．

どのアプリケーションにおいても，本手法で特定したデータ型は手作業で作成したものと同じであり，生成した VM のサイズも同じであった．フルセット VM と比較すると，本手法で生成されたインタプリタのサイズは，pimorse では約 70%，hygrothermograph では約 50%小さくすることができた．また，型推定を用いた方法を適用した場合よりも，本手法は不要な内部データ型を多く省くことができた．どちらのアプリケーションにおいても，本手法で生成したインタプリタのサイズは型推定を用いた場合よりも約 30%小さかった．

7.2.4　議論

手作業で作成したオペランド仕様を基に生成したインタプリタは，フルセット VM のインタプリタのサイズと比較して，半分以下であった．本手法を用いることの利点は，これに近いサイズのインタプリタを生成できることである．実際に本実験では，本手法で生成した VM は手作業で作成したオペランド仕様で生成した VM と同じサイズであった．本手法の欠点は，アプリケーションを網羅的に検査するユニットテストを必要とすることである．しかし，多くのアプリケーション開発でユニットテストは日常的に行われており，本手法はそのユニットテストをそのまま流用できるため負担は大きくないと考える．

型推定を用いた方法について，表 7.4 をから分かることは，VM 命令単位とほぼ同じであるということである．例えば，hygrothermograph の実行において，加算演算子は数値の足

し算と文字列の結合のためにしか使用されていない．しかし，型推定を用いた方法では，保守的に全ての内部データ型に対応しなければならず，VM 命令単位と同じ結果になった．このようになる理由は，hygrothermograph のプログラム中でオブジェクトのプロパティアクセスで得た値や関数の返り値を加算演算子に適用していることにある．JavaScript はこのような値のデータ型を静的に調べるのは困難であるため，必要な演算子の機能は保守的に見積もる必要がある．結果的に，図 7.3 で示したように，型推定を用いた方法を利用したときのインタプリタのサイズは，どのアプリケーションにおいても，VM 命令単位で生成した場合とほぼ同じになった．

第 8 章

おわりに

本研究では，カスタマイズした JavaScript VM のインタプリタを生成するための新しいアプローチを提案した．開発したフレームワーク eJSTK は，プログラマから与えられたデータ型仕様に基づいて VM 内部のデータ型の表現をカスタマイズする．生成された VM は，入力のデータ型仕様に合わせた，効率の良い型ディスパッチコードを自動生成する．さらに eJSTK は，各 VM 命令のオペランドに与えられ得るデータ型を定義したオペランド仕様を入力にとることで，対象のプログラムの実行に不要な VM 命令の機能を削った，小さな VM を生成することができる．

さらに，本研究ではオペランド仕様を作るためのプロファイラも開発した．このプロファイラは，プログラマ自身が作成したユニットテストを利用し，ユニットテストの実際の実行トレースから必要な VM 命令の機能を特定する．これによって，アプリケーションの実行に不要な VM 命令の機能を多く省くことができる．実際に組込みアプリケーションに適用したところ，VM のうち，インタプリタ部分のサイズを約 50〜70 %省くことができた．

謝辞

本研究を行うにあたり，丁寧なご指導をいただいた高知工科大学の鵜川始陽准教授と電気通信大学の岩崎英哉教授に深く感謝致します．また，高知工科大学の松崎公紀教授と高田喜朗准教授，東京大学の佐藤重幸先生からは多くのアドバイスを頂きました．ありがとうございました．研究室メンバをはじめ，支えてくださった全ての人に感謝致します．

参考文献

[1] Philip Koopman. *Better Embedded System Software*. Drumnadrochit Press, 2010.

[2] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. Efetch: Optimizing instruction fetch for event-driven webapplications. In *Proc. 23rd International Conference on Parallel Architectures and Compilation*, PACT 2014, pp. 75–86, 2014.

[3] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural implications of event-driven server-side web applications. In *Proc. 48th International Symposium on Microarchitecture*, MICRO-48, pp. 762–774, 2015.

[4] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, Vol. 35, No. 2, pp. 97–113, June 2003.

[5] Takafumi Kataoka, Tomoharu Ugawa, and Hideya Iwasaki. A Framework for Constructing JavasSript Virtual Machines with Customized Datatype Representations. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC 2018, pp. 1238–1247, New York, NY, USA, 2018. ACM.

[6] Tomoharu Ugawa, Hideya Iwasaki, and Kataoka Takafumi. eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems. *Journal of Computer Languages (to appear).*

[7] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen - a generator of efficient virtual machine interpreters. *Softw., Pract. Exper.*, Vol. 32, No. 3, pp. 265–294, 2002.

[8] David Gregg and M. Anton Ertl. A language and tool for generating efficient virtual machine interpreters. In *Proc. Domain-Specific Program Generation*, pp. 196–215, 2003.

[9] M. Anton Ertl. A portable forth engine. In *Proc. EuroFORTH '93 Conference*,

1993.

[10] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an efficient Java interpreter. In *Proc. High-Performance Computing and Networking, 9th International Conference*, HPCN Europe 2001, pp. 613–620, 2001.

[11] Andrew Beatty, Kevin Casey, David Gregg, and Andrew Nisbet. An optimized Java interpreter for connected devices and embedded systems. In *Proc. 2003 ACM Symposium on Applied Computing*, SAC 2003, pp. 692–697, 2003.

[12] Kazuaki Tanaka, Avinash Dev Nagumanthri, and Yukihiro Matsumoto. mruby – rapid software development for embedded systems. In *Proc. 15th International Conference on Computational Science and Its Applications*, ICCSA 2015, pp. 27–32, 2015.

[13] Takuya Azumi, Yuki Nagahara, Hiroshi Oyama, and Nobuhiko Nishio. mruby on TECS: component-based framework for running script program. In *Proc. IEEE 18th International Symposium on Real-Time Distributed Computing*, ISORC 2015, pp. 252–259, 2015.

[14] Stefan Brunthaler. Inline caching meets quickening. In *Proc. Object-Oriented Programming, 24th European Conference*, ECOOP 2010, pp. 429–451, 2010.

[15] Stefan Brunthaler. Efficient interpretation using quickening. In *Proc. 6th Symposium on Dynamic Languages*, DLS 2010, pp. 1–14, 2010.

[16] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1995, pp. 322–332, 1995.

[17] Gregory B. Prokopski and Clark Verbrugge. Analyzing the performance of code-copying virtual machines. In *Proc. 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2008, pp. 403–422, 2008.

[18] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proc. 8th Symposium on Dynamic Languages*, DLS 2012, pp. 73–82, 2012.

[19] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing ast interpreters. In *Proc. 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pp. 123–132, 2014.

[20] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language. *SIGPLAN Not.*, Vol. 24, No. 7, pp. 146–160, June 1989.

[21] Gem Dot, Alejandro Martínez, and Antonio González. Removing checks in dynamically typed languages through efficient profiling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pp. 257–268, Piscataway, NJ, USA, 2017. IEEE Press.

[22] Michael Haupt, Michael Perscheid, and Robert Hirschfeld. Type Harvesting: A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC 2011, pp. 1282–1289, New York, NY, USA, 2011. ACM.

[23] Manuel Serrano. JavaScript AOT Compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pp. 50–63, New York, NY, USA, 2018. ACM.

[24] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*, pp. 513–528, May 2010.

[25] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proceedings of the 22Nd ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 449–459, New York, NY, USA, 2014. ACM.

[26] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pp. 488–498, New York, NY, USA, 2013. ACM.

[27] 谷田英生, Li Guodong, Ghosh Indradeep, 上原忠弘. 記号実行エンジンを用いたJavaScript プログラムの単体テスト自動生成実行. ソフトウェアエンジニアリングシンポジウム 2014 論文集, 第 2014 巻, pp. 158–163, aug 2014.

[28] Google. V8 JavaScript engine, `https://developers.google.com/v8/`, 2015.

[29] Mozilla. Rhino, `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino`, 2016.

[30] Mozilla. Spidermonkey internals, `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals`, 2017.

[31] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* 5.1 edition, June 2011.

[32] James R. Bell. Threaded code. *Commun. ACM*, Vol. 16, No. 6, pp. 370–372, 1973.

[33] WebKit. Sunspider JavaScript benchmark, `https://webkit.org/perf/sunspider/sunspider.html`, 2015.

[34] JerryScript, `http://jerryscript.net/`, 2017.

[35] V7: Embedded JavaScript engine, `https://github.com/cesanta/v7`, 2016.