eJSVMにおける対話的なプログラミング環境の実現

大林 健造[1,a)]　岩崎 英哉[2,b)]

概要：近年，ネットワークに接続された機器を利用してデータの収集・制御を行う IoT (Internet of Things) 技術が利用される機会が増えてきている．eJS (embedded JavaScript) プロジェクトでは，IoT デバイスにおけるプログラム開発を，広く用いられている高級言語である JavaScript で行い，アプリケーション開発の効率化を目指している．eJS プロジェクトでは，データ型や命令などの JavaScript の一部機能をアプリケーション毎に取捨選択することで，計算資源の乏しい IoT デバイスでも動作するオーダーメイド JavaScript 処理系を実現している．eJS の仮想機械 (eJSVM) は JavaScript のプログラムを仮想機械命令列にコンパイルして実行するため，IoT デバイス向けのプログラム開発におけるデバッグ時など，コード変更・実行の頻度が高い時には，毎回プログラムをコンパイルしなければならず，動作確認が手間になってしまう．そこで，eJSVM において Lisp のような対話的なプログラム実行環境を実現する．それによって，プログラムデバッグ時のターンアラウンド時間を削減することができ，プログラム開発の効率化が期待される．

キーワード：REPL，組み込みシステム，JavaScript，仮想機械

1. はじめに

近年，ネットワークに接続された機器を利用してデータの収集・制御を行う IoT (Internet of Things) 技術が注目されている．IoT に利用される機器は IoT デバイスと呼ばれる．

IoT デバイス内の計算機は一般的に低コストなものが使われるため，主記憶，CPU 能力など，利用できる計算資源は限られている．こういった限られた計算資源でも動作するプログラムの開発においては，アセンブリ言語や C 言語などが用いられていることが多い．しかし，これらの言語を用いてメモリ管理などを意識しながらプログラム開発を行うのは，コードの見通しが悪くなり，開発者の負担となってしまう．

我々は，IoT デバイスを始めとする組み込みシステムにおけるプログラム開発を，広く用いられている高級言語である JavaScript で行うことにより，アプリケーション開発を効率化することを目指し，組み込みシステム用の JavaScript 処理系を開発している．我々はこれを eJS(embedded JavaScript) [1] と呼んでいる．組み込みシステムにおけるプログラム開発に JavaScript を利用できれば，メモリ管理などを意識する必要がなくなり，従来のアセンブリ言語や C 言語でのプログラム開発と比較して，開発者の負担が軽減されることが期待される．

eJS では JavaScript のコードを専用のコンパイ

[1] 電気通信大学情報理工学部情報・通信工学科
[2] 電気通信大学大学院情報理工学研究科
[a)] obayashi@ipl.cs.uec.ac.jp
[b)] iwasaki@cs.uec.ac.jp

ラ (以下，eJSC と書く) でコンパイルし，出力された命令列を仮想機械 (以下，eJSVM と書く) 上で実行する．eJS 上で動作するプログラム開発を行う際，記述した JavaScript のプログラムの実行結果を得るためには eJSC の起動・eJSVM の実行を手動で行う必要がある．デバッグ時など JavaScript コードの変更，実行結果の確認の頻度が高い時にはこれらの操作が手間になり，一般的にはターンアラウンド時間が長くなるため，効率的なプログラム開発が難しい．

そこで本稿では，eJSVM において動作する対話的なプログラム実行環境 (Read-Eval-Print Loop，以下，REPL と書く) を実現する．これによって，デバッグ時など，コード変更と実行の頻度が高い時にも動作確認を手間なく行えるようになり，組み込みシステムにおけるアプリケーション開発がより効率的に行えるようになることが期待される．

本稿の構成は以下の通りである．2 章では eJS の概要と実行環境を説明する．次に，3 章で REPL の概要と設計について，4 章で REPL を構成するプログラムの実装について述べる．5 章では組み込みシステム向けプログラム開発の関連研究について述べ，6 章で今後の課題と本稿のまとめを述べる．

2. eJS プロジェクト

eJS (embedded JavaScript) プロジェクトとは，組み込みシステムにおけるプログラム開発を広く用いられている高級言語である JavaScript で行い，見通しよいプログラム開発を可能にし，アプリケーション開発の効率化・試作の容易化を目指すものである．eJS プロジェクトでは，データ型や命令などの JavaScript の一部機能をアプリケーション毎に取捨選択することで，計算資源の乏しい IoT デバイスでも動作するオーダーメイド JavaScript 処理系を実現している．

2.1 eJS の全体像

eJSVM は，組み込みシステム向けに開発している，JavaScript プログラム用のレジスタベース仮想機械である．eJSVM は，eJSC によって JavaScript プログラムからコンパイルされた，VM 命令列のインタプリタであり，VM 命令をひとつずつ解釈して実行していく．

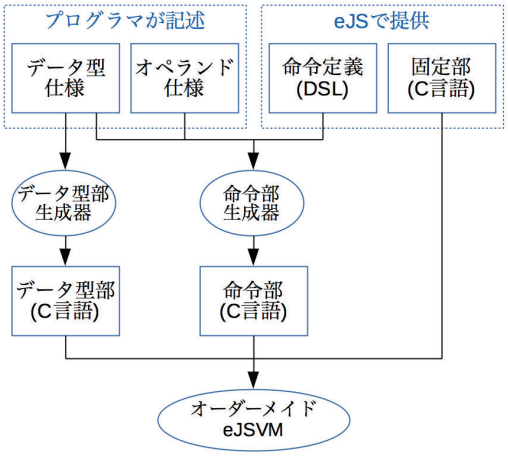

図 1 eJS の概観

JavaScript は多機能な言語であるが，組み込みシステムにおいて，その全ての機能が必要となることは稀である．例えば，“+” オペレータのようにオペランドとして複数の型 (数値や文字列) が渡されるような場合，VM は解釈の際にオペランドの型から適切な命令を判断して割り当てる必要がある．しかし，組み込みシステムにおいては「“+” オペレータに渡されるオペランドは必ず数値である」というものも存在すると考えられる．そのような場合，“+” オペレータに渡されるオペランドは数値であることが分かっているので，“+” オペレータに対応する命令は数値の加算命令であることが分かり，命令ディスパッチをより効率的に行える．このように eJS では，データ型やオペランドの仕様をプログラマが指定し，JavaScript のサブセットを eJSVM で扱うようにすることで，VM のサイズは小さくなり，組み込みシステムに向いた実装を行えるようになっている．

eJSVM は，図 1 のように，プログラマによって指定されたデータ型とオペランドの仕様と eJS で提供する DSL で記述された命令定義および VM の基本部分である固定部から自動生成される．データ型とオペランドの仕様は，主に eJSVM 内で用

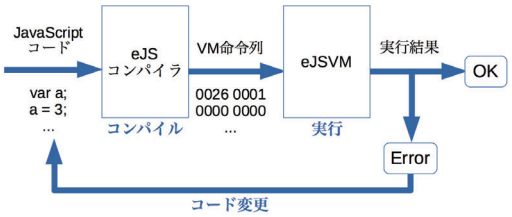

図 2　eJSVM 実行までの流れ

```
1  /* ヘッダ*/
2  0001 --- number of functions
3
4  /* 関数情報*/
5  0000 --- call_entry
6  0000 --- send_entry
7  0000 --- n_locals
8  0009 --- n_insns
9
10 /* 命令*/
11 0026 0001 0000 0000 --- getglobalobj 1
12 0022 000a 0000 0000 --- setfl 10
13 0001 0002 0000 0016 --- specconst 2
       undefined
14 0002 0004 0000 0006 --- string 4 "a"
15 0018 0003 0004 0000 --- getglobal 3 4
16 0000 0005 0000 0002 --- fixnum 5 2
17 0007 0002 0003 0005 --- mul 2 3 5
18 0023 0002 0000 0000 --- seta 2
19 0028 0000 0000 0000 --- ret
20
21 /* リテラル*/
22 6100 --- STRING "a"
```

図 3　VM 命令列の内容

いるデータ表現や各オペレータに渡されるオペランドの種類を記述する．

2.2　eJSVM の実行環境

JavaScript のコードを eJSVM 上で実行するまでの流れを図 2 に示す．

プログラマはまず JavaScript コードを記述し，eJSC に入力する．eJSC は JavaScript コードを VM 命令列にコンパイルして出力する．この VM 命令列は，eJSVM が解釈するために設計された，eJS プロジェクト独自のものである．この VM 命令列を eJSVM にロードすることでプログラムを実行することができる．

```
eJSrepl> 2+3
it = number:5

eJSrepl> f=function(x){return x*(x+1)/2;}
it = object:object

eJSrepl> f(4)
it = number:10
```

図 4　入出力例

eJSC は VM 命令列をバイナリで出力する．図 3 に JavaScript コード「a*2」をコンパイルした結果のバイナリの内容と各部に対応する VM 命令を示す．

VM 命令列にはヘッダ，関数情報，命令，リテラルの 4 種の情報が含まれる．ヘッダには，図 3 の 1 行目のように，VM 命令列内の関数の総数が書き込まれている．関数情報には，図 3 の 3 行目から 6 行目のように，call，send の各命令について呼び出された時の命令開始位置 (call_entry，send_entry)，ローカル変数の総数 (n_locals)，命令の総数 (n_insns) などが書かれている．それに続いて，各関数の命令列が 1 命令あたり 8 バイトで記述されている．各命令は，先頭 2 バイトで各命令に対応する命令番号が表され，以後 2 バイトずつ最大 3 つの引数が続く形式になっている．

eJSVM は，eJSC が出力した VM 命令列を読み，ヘッダの関数総数をもとに，関数ごとに関数情報，命令，リテラルをロードする．ロードされた関数は配列によって管理され，ロードされた順に 0 から始まる関数番号が付与される．その後，一番初めにロードした関数 (番号 0 の関数) から実行を行う．

3.　対話的プログラム実行環境の設計

3.1　基本設計

eJSVM において，図 4 のような対話的なプログラム実行環境を実現する．ユーザは REPL のプロンプトに対して評価したい式を入力する．REPL

はその式を読み込んで評価し，結果の値を特別な大域変数 it に代入する．it の値は次の式で使うことができる．典型的な Lisp インタプリタのように，プログラムを直接解釈実行するインタプリタの場合には，インタプリタの内部に REPL を合わせて実装することが多い．しかし eJS の場合は，ユーザが記述した JavaScript プログラムを解釈実行するのではなく，VM 命令列にコンパイルした後に VM 命令列を解釈実行するという実行形態になっている．したがって eJSVM の REPL では，ユーザが与えた JavaScript の式を毎回 VM 命令列にコンパイルし，その命令列を実行して結果を表示するように設計する．

REPL 実装の概略図を図 5 に示す．REPL を実装するには，以下の仕組みが必要である．

(1) ユーザが入力した JavaScript コードを受け取る部分
(2) JavaScript コードを VM 命令列にコンパイルする部分
(3) VM 命令列を eJSVM 本体に受け渡す部分
(4) VM 命令列を実行する部分
(5) 実行結果を表示する部分
(6) 以上の行程を繰り返し行う部分

(2)，(4) については既存の eJS の構成要素である eJSC と eJSVM を用いればよい．ただし，通常の eJS は 1 回の起動で複数のプログラムを連続して実行するようになっていない．さらに，図 3 のような VM 命令列は，コマンドライン引数で指定されたファイルを通して受け取れるようになっている．REPL の実装にあたって連続したコード読み込みをが可能になるよう一部改変を行う．

また，(1)，(3)，(5)，(6) については，eJSC および eJSVM とは独立して，REPL のフロントエンドが受け持つこととした．そうすることで，eJSC や eJSVM の変更が行われた場合も REPL の機能自体への影響を最小限に抑えることができる．

ユーザはこのフロントエンドのプログラムを介して JavaScript コードの入力を行い，実行結果の出力を得るようにする．

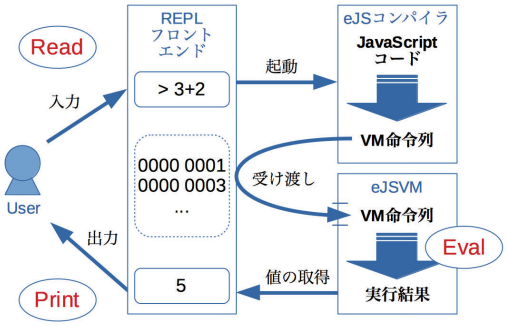

図 5　REPL 実装概略図

3.2　連続したコード読み込み

eJSVM は VM 命令列のファイルをただひとつ読み込み，その命令を実行するようになっている．まずこの部分を VM 命令列読み込みが連続してできるように改造する．方法としては，eJSVM が読み込んだコードを実行し終えたら結果を出力し，再びコード読み込み待機状態になるように，コード読み込みと実行をループとして実装する．

また，連続してコードを読み込む時，関数は今までロードされてきた関数の配列の続きにロードされ，実行を開始するべき関数の関数番号は実行のたびにずれていく．eJSVM は関数番号 0 番から実行を開始するようになっているので，この部分にも手を入れ，実行のたびに初めに実行する関数の関数番号がずれていくように実装を行う．eJSC においても，初めに実行する関数の関数番号を任意に変更できるようなオプションを実装する．

3.3　REPL フロントエンド

続いて，ユーザと eJSVM の間に入り，入出力データの受け渡しなどの役割を担う，REPL のフロントエンド部の実装を行う．ユーザが JavaScript コードを入力すると，フロントエンド部は eJS コンパイラを起動し，入力されたコードに対応する VM 命令列を得る．これを eJSVM に受け渡し，eJSVM に実行させ，そこから出力される実行結果を eJSVM から受け取り，フロントエンド部で表示する．

実装の方法としては，まず REPL フロントエンド起動時に，eJSVM も同時に起動し，コード読み

込み待機状態にする．ユーザのコード入力に応じて REPL フロントエンドは eJSC を起動しコードのコンパイルを行い，VM 命令列ファイルを得る．このファイルの内容を読み，eJSVM に送って実行させ，その実行結果を表示する．

4. REPL の実装

4.1 連続コード読み込み

eJS において REPL を実装するうえで，まず eJSVM が連続したコード読み込みを可能にする必要がある．コードを連続して読むことで，eJSVM は関数を次々ロードしていくため，実行を開始する関数の関数番号は，必ずしも 0 ではない．そのため，eJSVM および eJSC について，関数番号を参照する部分を改変する必要がある．

4.1.1 eJSVM のロード部

REPL において，eJSVM は，入力されたバイナリデータを読み，実行を行う，ということを繰り返し行う．これを，VM 命令列読み込み処理部分と実行部分を `while` ループによって繰り返し行うことで実装した．

また，REPL フロントエンドの実装のため，ファイルから VM 命令列を読み込むだけでなく，標準入力からバイナリデータを受け取ることも可能とした．

eJSVM の起動時にオプション「`-R`」をつけることで，連続でのコード読み込みを行うようにした．

4.1.2 eJSC の VM 命令列生成部

eJSC においても，起動時に eJSVM が既に読み込んでいる関数の総数をコマンドラインオプションを通して受け取り，生成する VM 命令列内の関数番号をその数だけずらすことができるようにした．

4.2 REPL フロントエンド

REPL フロントエンドの起動時に，子プロセスとして eJSVM をオプション「`-R`」で起動する．eJSVM に対するデータの入出力には `pipe()` を用いる．

REPL 実行において，REPL フロントエンドは次のように動作する．

(1) プロンプトを表示し，ユーザに JavaScript コードを入力させる．
(2) ユーザの入力した JavaScript コードを受け取り，それを一時的な JavaScript ファイルとして保存する．
(3) eJSC を `system` 関数で起動し，そのファイルを受け渡して VM 命令列のバイナリファイルを受け取る．
(4) バイナリファイルの内容を読み出し，eJSVM にパイプを経由して渡すことで，実行結果を得る．
(5) 実行結果を出力する．

5. 関連研究

本稿での REPL の実装は，eJSC および eJSVM といった処理系と分離して REPL フロントエンドを実装し，コードを毎回 VM 命令列にコンパイルし VM で実行する方法で行った．

REPL が標準で提供されている言語としては，多くの Lisp 処理系，Haskell (GHCi)，Scala (標準の scala，sbt console)，Ruby (irb)，Smalltalk (gst) など，多岐にわたる．Haskell や Scala はそれぞれ，REPL 実行時には各コードを都度コンパイルしてから実行を行っているが，Haskell は REPL の機能をコンパイラ (GHC) 内で実行しており，Scala は REPL 用のフロントエンドをコンパイラとは別に持つ．本稿の実装は Scala と同様の方式をとっている．

また，Java に対する REPL 実装の例としては Allen ら [4] の実装が挙げられる．Allen らは，Java のインタプリタである DynamicJava [5] を拡張し，プログラミング教育のための Java の対話的なプログラミング環境として REPL の実装を提案している．この実装においては，DynamicJava の Java インタプリタ内に REPL の機能を実装しており，本稿での実装とは異なる．

6. おわりに

本稿では，組み込みシステム向けの JavaScript 処理系のひとつである eJS について，プログラムのデバッグ時などに手間のかかる実行結果確認の

負担を軽減する方法として，eJSVM 上で動作する REPL の実装を行った．本稿での実装では，従来の eJSC および eJSVM の変更を最小限に抑え，REPL の機能のほとんどを REPL フロントエンド部で実装した．そのため，eJSVM のアップデートなどにおいて，REPL のための機能を変更する必要は最小限に抑えられている．

本稿での実装にはまだ改善すべき余地がある．そのひとつとして REPL に伴う VM 命令列保持領域のごみ集めがある．eJSVM は VM 命令列を読むと関数情報をもとに，各関数を配列にロードし，VM 命令を保持する領域を確保する．本稿で実装した REPL においては，読み込んだ個々の JavaScript コード片をそれぞれ小さな関数としてコンパイルし eJSVM に読み込むようになっている．したがって，REPL でコードを実行するたび VM 命令列保持領域は少しずつ消費されていってしまう．しかし，REPL においては一時的な計算にのみ使われ，保持される必要のない関数および VM 命令列が多く存在すると考えられる．このような命令列に対応する命令コード領域は結果の出力が終われば解放してしまっても問題ない．VM 命令保持領域の消費を最低限に抑えるために，このような保持する必要のない命令コード領域をピープ内にとるようにし，通常のごみ集めにより不要な VM 命令列保持領域を解放するように改善していく予定である．

参考文献

[1] T. Kataoka, T. Ugawa, and H. Iwasaki.: A Framework for Constructing JavaScript Virtual Machines with Customized Datatype Representations, *The 33rd ACM/SIGAPP Symposium On Applied Computing*, SAC 2018, pp. 1238-1247, (2018).

[2] E. Sandewall.: Programming in an Interactive Environment: the "Lisp" Experience, *ACM Computing Surveys*, 10(1), pp. 35-71 (1978).

[3] E. Allen, R. Cartwright, and B. Stoler.: DrJava: A Lightweight Pedagogic Environment for Java, *SIGCSE '02 Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pp. 137-141 (2002).