修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学研究科　　情報・ネットワーク工学専攻　博士前期課程		
氏　　　名	平澤 祐太	学籍番号	2031122
論 文 題 目	JavaScript 仮想機械を記述するための領域特化言語		

要　　旨

JavaScript などの動的なマネージド言語の仮想機械処理系 (VM) の実装は，典型的には C や C++ が用いられる．動的な言語の VM は，実行時のデータ型に応じた分岐処理を多く持つため，C や C++ で処理を直接記述する場合，あらゆる型の組み合わせに関して記述する必要があり，記述が非常に煩雑になる．一方で，組み込み機器上では特定のアプリケーションを繰り返し動作させると考えられるため，VM は限定されたデータ型に関する処理のみを持てばよい．しかし，C や C++ ではそのような VM を構築することは難しい．また，VM 内部のデータの型は実行時に決定するため，C コンパイラは VM 内部のデータ型に関するエラー検出やコンパイル時最適化ができない．他にも，GC をはじめとする VM の様々な処理を手動で実装するにはプログラマへの負担が大きいという問題がある．本論文では，以上の問題点を解決するために，VM の実装を記述するための独自の領域特化言語である VMDL を設計し，組み込みシステム向け JavaScript 処理系の実装に使用した．VMDL は，VM 内部のデータ型粒度で解析可能な型システム，VM の実装に沿ったコードを自動生成できるよう関数にアノテーションを与えられる仕組み，多変数の型ディスパッチを簡潔に記述できるという特徴を持っている．その結果，C 言語による実装では検出できなかったバグを検出したり，VM 内部のデータ型粒度で最適化したりすることができた．また，ベンチマークを用いた評価を行い，既存研究による実装と比べて，VM サイズ・実行速度の両面からほぼ同等，あるいは結果が改善することを確認し，VMDL の有効性を示した．

令和 3 年度修士論文

JavaScript 仮想機械を記述するための領域特化言語

電気通信大学 情報理工学研究科
情報・ネットワーク工学専攻
コンピュータサイエンスコース

学籍番号	:	2031122
氏名	:	平澤 祐太
主任指導教員	:	岩崎 英哉
指導教員	:	寺田 実
提出日	:	2022 年 1 月 28 日

要旨

JavaScript などの動的なマネージド言語の仮想機械処理系（VM）の実装は，典型的にはC や C++ が用いられる．動的な言語の VM は，実行時のデータ型に応じた分岐処理を多く持つため，C や C++ で処理を直接記述する場合，あらゆる型の組み合わせに関して記述する必要があり，記述が非常に煩雑になる．一方で，組み込み機器上では特定のアプリケーションを繰り返し動作させると考えられるため，VM は限定されたデータ型に関する処理のみを持てばよい．しかし，C や C++ ではそのような VM を構築することは難しい．また，VM 内部のデータの型は実行時に決定するため，C コンパイラは VM 内部のデータ型に関するエラー検出やコンパイル時最適化ができない．他にも，GC をはじめとする VM の様々な処理を手動で実装するにはプログラマへの負担が大きいという問題がある．本論文では，以上の問題点を解決するために，VM の実装を記述するための独自の領域特化言語である VMDL を設計し，組み込みシステム向け JavaScript 処理系の実装に使用した．VMDL は，VM 内部のデータ型粒度で解析可能な型システム，VM の実装に沿ったコードを自動生成できるよう関数にアノテーションを与えられる仕組み，多変数の型ディスパッチを簡潔に記述できるという特徴を持っている．その結果，C 言語による実装では検出できなかったバグを検出したり，VM 内部のデータ型粒度で最適化したりすることができた．また，ベンチマークを用いた評価を行い，既存研究による実装と比べて，VM サイズ・実行速度の両面からほぼ同等，あるいは結果が改善することを確認し，VMDL の有効性を示した．

目次

1 はじめに 1
1.1 プログラムの記述性 1
1.2 プログラムの安全性 2
1.3 VM サイズ 2
1.4 本論文の目的と構成 3

2 eJS プロジェクト 4
2.1 概要 4
2.2 VM の小型化のアイデア 4
2.3 命令の仕様 5
2.4 eJS Toolkit 6
2.5 従来の eJSVM 開発 7

3 VMDL 10
3.1 概要 10
3.2 構文 10
3.3 データ型 14
3.4 型付け規則 15
3.5 VMDL 関数の記述例 19

4 VMDLC 24
4.1 概要 24
4.2 型解析処理 24
4.3 型変換関数スペックの生成 27
4.4 関数アノテーションの処理 28
4.5 最適化処理 33
4.6 C コードの生成 35

5 不要機能の削減機構とプロファイラ 40
5.1 コードセレクタ 40
5.2 プロファイラ 40

6 評価 42

6.1 実験環境 42
6.2 記述性と安全性 43
6.3 効率性 45

7 関連研究 50

8 おわりに 52

謝辞 53

参考文献 54

1　はじめに

JavaScript [22] は，ウェブアプリケーションの用途に留まらず，サーバサイドのプログラムや組込みシステム向けのアプリケーションの開発にも使用されている．JavaScript を含む動的なマネージ言語の仮想機械 (Virtual Machine: VM) は典型的には C 言語や C++ （以降，これらをまとめて C 言語と呼ぶ.）を用いて実装される [24]．C 言語を用いる場合，プログラマは低レベルの処理を柔軟に記述でき，自由度が高いという利点がある．一方で，主に VM の開発者にとってふたつの，VM のユーザにとってひとつの問題がある．

1.1　プログラムの記述性

まず，開発者にとっての 1 点目の問題は，プログラムの記述性である．C 言語を用いて VM のコードを記述する場合，コードが複雑になってしまい記述性が低い．

たとえば，動的型付け言語では式の型が動的に決定するため，実行時にデータ型による分岐処理（以降，型ディスパッチ処理と呼ぶ.）が多数発生する．C 言語で型ディスパッチ処理を記述する場合，典型的には `switch` 文を組み合わせて記述する．しかし，特に多変数に基づく型ディスパッチを行う場合に `switch` 文をネストさせたコードとなり，コードの見通しが悪くなってしまう．加えて，JavaScript の VM は型ディスパッチ処理の後，データを別の型へ変換したのち再び型ディスパッチ処理を行う場合もあるため，コードがより複雑になってしまう．

他の例として，VM が持つ機能のための特定のコード片の挿入をしなければならない場合がある．そのひとつに，ごみ集め (Garbage Collection: GC) のための，生存オブジェクトのルート集合 (GC ルート) の管理やライトバリアの記述などがある．GC 機構が正しく動作するためには，GC ルートを正しく管理しなければならない．そのため，プログラマはコード中の GC を起こしうる関数呼び出しの前で，呼び出し後も使用する局所変数のうち GC 対象の値を保持するものを GC ルートへ退避し，関数呼び出し後に復旧させなければならない．例えば，2 つの変数 `a` と `b` が GC 対象の値を持っており，関数 `g` が GC を起こしうるとする．関数 `g` を呼び出す場合，その前後に GC ルートへ変数のアドレスを登録する操作 `push` と取り出す操作 `pop` を行うコード片を次のように挿入する．

```
push(&a); push(&b); g(...); pop(); pop();
```

もしも `push` を行わなければ，関数 `g` 内で GC が発生したとき変数 `a` と `b` はダングリングポインタとなってしまう．そのようなコーディング上のミスは，VM 開発者が注意深くコーディングしていても発生しやすく，バグの原因となってしまう．更に，ミスをしていても関数 `g` 内で GC が発生しなければバグが露見せず，また，コンパイラはこの問題を静的に発見することができな

い．これについては 1.2 節で述べる安全性の面においても大きな問題である．

1.2　プログラムの安全性

次に，開発者にとって 2 点目の問題はプログラムの安全性である．C 言語を用いてコードを記述する場合，データ型に関するバグをコンパイル時に検出できない場合がある．

典型的な VM の実装において，対象言語の第一級のオブジェクトのデータ型（以降，VM 内部データ型，あるいは単に内部データ型と呼ぶ）すべてを表す C のデータ型（ここでは `Value` 型とする）を定義する場合がある．`Value` 型はその値の一部に内部データ型を表すタグを持たせることでデータ型を判別することが，多くの実装で用いられている．動的型付け言語の場合，内部データ型は実行時に決定するため，`Value` 型の変数を引数に取る関数が特定の内部データ型を期待する場合，典型的には関数の先頭で `assert` マクロなどを用いた検査を行う．例えば，関数 `f` が実行時に `Fixnum` という内部データ型を引数に期待している場合，C コード上で `f` は以下のように定義されるであろう．

```
void f(Value a) { assert(isFixnum(a)); ... }
```

ここで，`isFixnum` は引数に与えられた `Value` 型の値が `Fixnum` 型の値であるかを返却する関数とする．このような `assert` を用いた検査は実行時に行われるため，関数 `f` が `Fixnum` 型以外の内部データ型の値で呼び出されていないことをコンパイル時に検査することはできない．

他の例として，`Value` 型と C 言語のデータ型の変換に関するバグをコンパイル時に検出できないという問題がある．例えば，`Value` 型が `uint64_t` 型の型シノニムとして定義されており，`Value` 型の変数 `n` は，下位 3 ビットが型を表すタグであり，上位 61 ビットがその変数の持つデータ本体だとする．`n` が `Fixnum` 型の値を保持し， `Fixnum` ではデータ本体部に即値が格納されているとすれば，以下のように 3 ビット右シフトをすることでデータ本体を取り出すことができる．

```
int64_t x = (int64_t)n >> 3;
```

しかし，次のように右シフト操作を忘れていたとしても，C コンパイラは型エラーとして報告しない．

```
int64_t x = (int64_t) n;
```

結果として，開発者はこのようなバグを見落としてしまう可能性がある．

1.3　VM サイズ

最後に，利用者にとっての問題は VM サイズである．

一般に，組込みシステム上で動作するアプリケーションは，温度や湿度のデータを測定してサーバに送信するなど，特定の目的を持っている．さらに，同じアプリケーションが繰り返し実行されることが多い．そのため，組込みシステム向けの VM は対象言語が持つ機能すべてを提供する必要はなく，対象アプリケーションにとって必要最小限の機能がありさえすれば十分である．

組み込みシステムが持つ主記憶等の計算資源は限られているため，VM サイズを小さくすることは重要である．しかし，C 言語で記述された VM コードでは，型ディスパッチ処理に `switch` 文が使用され，対象アプリケーションにとって不要な分岐が含まれてしまう．更に，対象のアプリケーションによっては，一部の命令や組み込み関数そのものが不要になる可能性がある．しかし，そのような不要コードを特定して削減することは難しく，VM サイズ削減を妨げてしまう．

1.4 本論文の目的と構成

本論文では，以上の問題点を解決するために，C 言語に代わって組み込み機器向けの VM コードをよりよく記述するための独自の領域特化言語（DSL）とそのコンパイラを設計・開発する．本 DSL は，C 言語では `Value` 型のように単一の C データ型で表現していた対象言語のデータ型を静的に型解析できるように設計されている．開発者はこの DSL を用いて VM の命令や組み込み関数などを記述する．DSL のコンパイラはそれらを静的に型解析して必要なコード部分を選択し，さらに型の情報を用いてより効率的な C コードへ変換する．

このようなアイデアに基づき，本研究では組込みシステム向けの JavaScript VM である eJSVM (embedded JavaScript VM) [13][20] の C ソースコードを生成するためのフレームワークを開発した．フレームワークに含まれるのは，eJSVM を記述するための静的型付けの DSL である VMDL (Virtual Machine Description Language) とそのコンパイラ VMDLC (VMDL Compiler)，および eJSVM の生成を支援するユーティリティツール群である．

本研究で提案する VMDL は，C 言語と同等の記述力を確保することを目的としておらず，型ディスパッチ処理を効率的に記述する点に注目しており，ポインタに関する低レベル操作や使用頻度の低い操作などを記述するための機能は提供していない．そのような操作は C 言語を用いて記述し，必要があれば VMDL 側から呼び出す形で実装される．また，VMDLC は厳密に最適化されたアセンブリコードの生成も目指しておらず，一般的な最適化は C コンパイラに委ねている．

本論文の構成は以下の通りである．2 章では eJS プロジェクトの概要と目的，提案フレームワークについて述べる．3 章では本研究で提案する DSL について記述例も併せて述べる．4 章では DSL のコンパイラについて，型解析処理と各種最適化手法を述べる．5 章ではユーティリティツールとして命令の仕様を生成するためのプロファイラについて述べる．6 章では評価について述べる．7 章では関連研究について触れ，8 章で結論を述べる．

2 eJS プロジェクト

2.1 概要

eJS プロジェクト [13][20] は，IoT デバイスなどの組み込み機器上で動作するアプリケーションの開発を，高級言語である JavaScript を用いて行うことを可能とするフレームワークの提供を目指している．一般的には，計算資源の乏しい組み込み機器上で動作するアプリケーションは，アセンブリ言語や C 言語を用いて開発されることが多い．しかし，これらの言語を用いる場合，プログラマが自身でメモリ管理を行う必要があるなど，プログラマに大きな負担がかかってしまう．開発を JavaScript で行うことができれば，プログラマはメモリ管理を自分で行う必要がなく，また，オブジェクト指向プログラミングやイベント駆動プログラミングなどの概念を用いて効率的にプログラムを記述することができる．

eJS が提供するフレームワークを eJS Toolkit (eJSTK) [20] と呼ぶ．eJSTK は，対象アプリケーション毎に eJSVM と eJSC を生成する．eJSVM は， eJS が目指す，組み込み機器向けの軽量で効率的な JavaScript VM である．組み込み機器上で動作するアプリケーションは，機器上で固定されて繰り返し動作し，かつ JavaScript が持つすべての機能を必要することが少ない．eJS では，この点に注目し，eJSVM に対象アプリケーションにとって必要最小限の機能だけを持たせることで，軽量で効率的な VM を実現する．

eJSVM はレジスタベースの VM で，eJS の研究のために設計された独自の命令セットを持っている．eJSVM 内部で使用される，JavaScript における第一級のデータ型（以降，JS データ型と呼ぶ）に対応する型を VM 内部データ型と呼ぶ．JS データ型と VM 内部データ型の対応を表 2.1 に示す．

eJSC は，JavaScript のソースコードを eJSVM の VM 命令列（バイトコード）へ変換するコンパイラである．eJSVM は対象アプリケーション毎に異なる仕様を持つため，同時に生成される eJSC とセットで使用することになる．なお，eJSC の詳細な実装は本論文の範囲ではないため，ここでは省略する．

2.2 VM の小型化のアイデア

JavaScript が持つ機能のひとつに，演算子のオーバーロードがある．表 2.2 に，JavaScript における加算演算子の型の関係を示す．JavaScript では，加算演算子は被演算子の型によって処理内容が異なる．数値同士の加算は数の加算を行い，文字列同士の場合は文字列を結合する．文字列と数値が与えられた場合には，数値を文字列表現に変換したのち，文字列を結合する．このように，Javascript の演算子は被演算子の型の組み合わせに応じた処理を行う必要がある．そ

表 2.1　VM 内部データ型

JS データ型	VM 内部データ型
Undefined, Null, Boolean	`Special`
String	`String`
Number	`Fixnum`, `Flonum`
Object	`Simple_object`
Array	`Array`
Function	`Function`, `Builtin`
Regexp	`Regexp`

表 2.2　JavaScript における加算演算子（一部）

		処理内容			例.
数値	+	数値	→	数値	`1 + 1`
文字列	+	文字列	→	文字列	`"abc" + "def"`
文字列	+	数値	→	文字列	`"abc" + 1`
文字列	+	真偽値	→	文字列	`"abc" + true`

のため，eJSVM の加算演算子に対応する命令は，オペランドに対する型ディスパッチ処理を内部で行う．一方で，eJS が対象とする組み込み機器上で動作するアプリケーションは，すべての組み合わせの処理を必要とすることは少ない．加算演算子の例であれば，数値計算を行うアプリケーションは，数値同士の加算は必要であるが，文字列同士など他の型の組み合わせの加算は必要ないであろう．そのような不要な型の組み合わせに関する処理のための VM 中のコードを削減することで，コンパクトで効率的な VM を実現する．

2.3　命令の仕様

eJS では，命令の扱うデータ型の取捨選択を，命令スペックという形式でユーザが指定する．命令スペックは，次の形式で与える．

$instruction$（$operand_1$, $operand_2$, ...）$action$

ここで，$instruction$ は VM 命令名，$operand_i$ はその命令の第 i オペランドであり，表 2.1 の右側に示す VM の内部データ型か，すべてのデータ型を表す記号 “_” あるいは入力でないオペランドを表す記号 “-” を指定する．アクションは，「指定されたデータ型を命令が受け取ること

許容する」ことを表す“accept”，「指定されたデータ型を命令は受け取ることを想定せず，受け取った際の挙動は未定義とする」ことを表す“unspecified”，「指定されたデータ型を命令は受け取ることを想定せず，受け取った際はエラーメッセージを出力する」ことを表す“error”のいずれかを指定する．この中で，“error”は開発時のデバッグのために用意されており，この指定をしたデータ型に関してエラーが発生しないことを確認したのち，“unspecified”に書き換えることが想定されている．

例えば，2 つのオペランドの加算を行う add 命令に関して，JS データ型における Number 型のみを許容する場合，次のように記述する．

```
add (-,Fixnum,Fixnum) accept
add (-,Fixnum,Flonum) accept
add (-,Flonum,Fixnum) accept
add (-,Flonum,Flonum) accept
add (-,_,_) unspecified
```

add 命令の第一オペランドは入力ではなく結果の格納先であるため，“-”を指定している．add 命令の第二，第三オペランドについて，JS データ型における Number 型に対応する VM 内部データ型のすべての組み合わせを“accept”とすることで，それらのデータ型で命令を実行しうることを指示し，それ以外の組み合わせを“unspecified”とすることで，その他のデータ型で命令を実行しないことを指示する．

また，現在の eJSTK では，命令の他に組み込み関数や型変換関数も扱うデータ型の取捨選択も行うことができる．これらに関する取捨選択も命令スペックと同じ形式で与える．*instruction* が関数名に，$operand_i$ がその関数の第 i 引数に対応する．以降，命令スペックと区別するため，組み込み関数の指定を組み込み関数スペック，型変換関数の指定を型変換関数スペックと呼び，これら 3 つをまとめて単にスペックと呼ぶ．

2.4 eJS Toolkit

この節では，eJSVM の生成に注目した eJSTK の構成について述べる．図 2.1 に，eJSTK における eJSVM 生成の流れを簡略化したものを示す．eJS では，2.2 節で述べた小型化のアイデアを実現するために，型ディスパッチ処理に関係する VM コード部分を DSL である VMDL によって記述する．こうすることで，型ディスパッチ処理が発生しうるデータ型の情報に基づいて不要なコードを削減することができる．そのため，eJSVM は C 言語による部分と，VMDL による部分に分かれて記述されている．VMDL による記述の対象となるのは，VM 命令，組み込み関数，VM 内部で使用される型変換関数などである．対して，C 言語による記述部分は，VM の初期化処理，GC 機構，型定義といった低レベルの操作や型ディスパッチ処理に関係の無いものが含まれている．

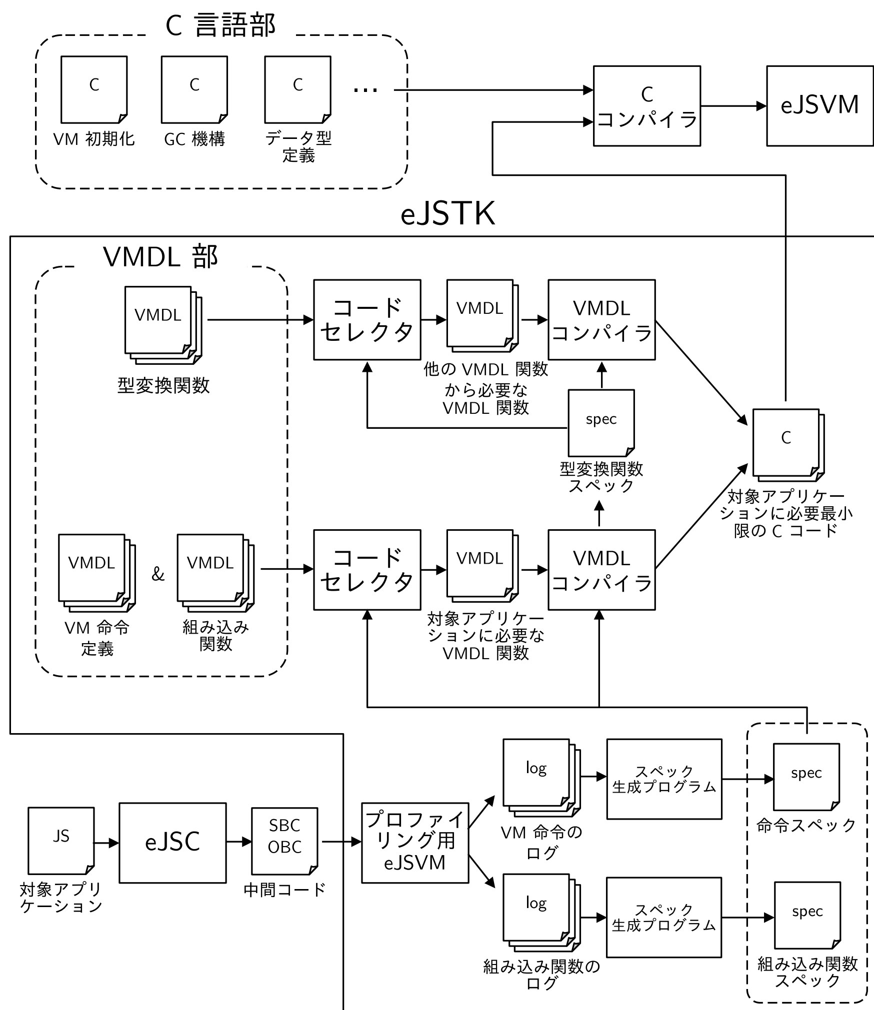


図 2.1　eJSVM に注目して簡略化した eJS Toolkit

スペックを用いて，型ディスパッチ処理が発生しうるデータ型の組み合わせを指定することにより，VMDL の処理系である VMDLC は VMDL コードから必要となる部分を選択して C 言語コード片を生成する．この C 言語コード片と C 言語による記述部分をあわせてコンパイルすることにより，最終的なカスタマイズ済みの eJSVM を得ることができる．

また，eJSTK は，対象アプリケーションに不要な機能を削減するコードセレクタと，スペック生成を支援するためのプロファイラを提供している．これらについては，第 5 章で詳しく述べる．

2.5　従来の eJSVM 開発

従来の eJSVM 開発においては，VMDL ではなく vmgen と呼ばれる DSL を用いて命令定義を記述していた．図 2.2 に，vmgen による `add` 命令の簡略化した記述例を示す．1 行目の `\inst` 節で `add` 命令を定義し，オペランドが結果を格納する `dst`，1 つめの入力オペランド `v1`，

```
1   \inst add (Register dst, Value v1, Value v2)
2
3   \when v1:fixnum && v2:fixnum \{
4     cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
5     dst = cint_to_number(context, s);
6   \}
7
8   \when v1:string && v2:string \{
9   DEFLABEL(STRSTR): __attribute__((unused));
10    dst = ejs_string_concat(context, v1, v2);
11  \}
12
13  \when v1:string && (v2:fixnum || v2:flonum || v2:special) \{
14    GC_PUSH(v1);
15    v2 = to_string(context, v2);
16    GC_POP(v1);
17    goto USELABEL(STRSTR);
18  \}
19
20  \when (v1:fixnum || v1:flonum) && (v1:fixnum || v1:flonum) \{
21    GC_PUSH(v2);
22    double x1 = to_double(context, v1);
23    GC_POP(v2);
24    double x2 = to_double(context, v2);
25    dst = double_to_number(context, x1 + x2);
26  \}
```

図 2.2　vmgen による add 命令定義（簡略版）

2 つめの入力オペランド `v2` の 3 つであることを表している．3 行目，8 行目，13 行目，20 行目の `\when` 節はそれぞれ型ディスパッチ処理の分岐を表している．`\when` と `\{` の間にオペランドの VM 内部データ型の条件件が記述されている．例えば，3 行目から始まる分岐の型条件は，`v1` が `Fixnum` 型であり，かつ `v2` も `Fixnum` 型であることを指定している．`\when` 節の `\{` と `\}` に挟まれた部分は C 言語によるコード片となっており，型条件に合致した場合にこのコード片の処理を実行する．例えば，`v1` と `v2` がともに `Fixnum` 型であった場合には 4 行目と 5 行目の処理を実行する．4 行目では，`v1` と `v2` を `Fixnum` 型の表現から，`cint` 型と呼ばれる C 言語上での整数表現に変換した後，加算して結果を変数 `s` に置く．5 行目で `s` を `Number` 型（`Fixnum` 型あるいは `Flonum` 型）に変換し，これをレジスタ `dst` に設定する．

2.5.1　問題点

vmgen は型ディスパッチ処理の内容を C 言語で直接記述しており，その中身を静的に解析するのは非常に煩雑である．そのため vmgen から C コードを生成する生成系はその中身の解析を行っていない．主としてこのことに起因して，vmgen を用いることには主に以下の 4 点の問

題がある．

まず 1 点目は，1.2 節で述べたように，int，double などの C 言語上で直接演算できる型（以降，C 言語データ型と呼ぶ）と VM 内部データ型を間違えた際に C コンパイラはコンパイル時に型エラーを報告しないことである．例えば，図 2.2 の 25 行目で C 言語データ型の変数 x1 と x2 の加算をしているが，ここで間違えて v1，v2 と書いてしまっても型エラーとはならない．v1，v2 は Value 型であるが，これは uint64_t 型の型シノニムであって，単なる整数型であるためである．そのためこの場合，問題のコードを含む処理が実行された後，正しくない結果が格納されたレジスタの値を使うまで，プログラムは見かけ上正しく動作してしまう．そのようなバグの発見と修正は難しく，プログラマにとって負担となってしまう．

2 点目は，1.1 節で述べたように，14 行目，16 行目にあるような GC のための処理を手動で挿入しなければならないことである．この箇所では，15 行目に呼び出される to_string 関数の中で GC が発生し得るため，その後 12 行目で使用する v2 を GC ルートに登録しなければならない．また，もし 9 行目，11 行目の処理を書き忘れたとしても，この箇所の処理が実行され，かつ GC が発生しない限りプログラムは正しく動作してしまう．そのため，GC に関係する処理の挿入し忘れによるバグは発見が難しく，これもプログラマにとって負担となってしまう．

3 点目は，VM 内部データ型に関する最適化を C コンパイラはできないことである．例えば，3 行目から始まる節は，型条件より v1 は VM 内部データ型の Fixnum 型である．そして 4 行目で fixnum_to_cint 関数を用いて cint 型の値へ型変換している．一方で，13 行目から始まる節の内部では 15 行目で変数を String 型の表現へ型変換している．ここでは型変換に to_string 関数を用いており，引数の VM 内部データ型を指定していない．そのため後者の場合，型変換対象は実行時に複数の VM 内部データ型を取りうるため， to_string 関数内で再び型ディスパッチ処理を行ってから型変換をしている．それに対して前者の場合，すでに変換対象は Fixnum 型しか取りえないことがわかっているため，fixnum_to_cint 関数を用いて直接型変換をしている．このように，変数が取りうる VM 内部データ型の文脈によっては適切な関数を選択することで，冗長な処理を省く最適化が可能である．しかし，このような最適化を C コンパイラはすることができないため，VM 開発者が手動で最適化したコードを記述する必要があり，負担となってしまう．

最後に 4 点目は，vmgen は基本的に VM 命令定義を記述するために設計されているため，VM が持つ型変換関数や組み込み関数の型ディスパッチ処理を記述できない．そのため，それらについては不要な機能を削減できない．さらに，vmgen による型ディスパッチ処理の記述は，処理の先頭で分岐する形にのみ対応しており，型ディスパッチ処理の中に別の型ディスパッチ処理を記述したり，型ディスパッチ処理そのものを複数個記述したりすることができない．そのため，型ディスパッチ処理の削減対象が VM 命令に限定されてしまい，組み込み関数などに対応することができない．

3 VMDL

3.1 概要

本論文は，2.5.1 節で述べた問題を解決するために新しい DSL である VMDL (Vitrual Machine Description Language) を提案する．VMDL は C 言語や vmgen に代わって VM を記述するための静的型付けの DSL である．その目的は，型ディスパッチ処理を静的型解析可能な形で導入し，スペックで指定されたデータ型に応じて不要なコードを削減することと，vmgen でプログラマの負担となっていた点について，より効率的かつ安全に記述できるようにすることである．

VM を記述する上で，VMDL は C 言語に比べて以下の利点を持っている．

記述性 命令や組み込み関数に関する VM 内部の詳細な実装を気にしなくてよい．関数アノテーションを用いて簡潔に記述できる．

安全性 C 言語レベルではコンパイル時に検出不可能な型エラーを検出可能な型システムを持つ．また，GC に関する操作を自動化できる．

効率性 C 言語レベルでは不可能な，VM 内部データ型に関する最適化ができる．

VMDL では命令定義も関数の形で記述する．また，VMDL は，外部の C 言語で記述されたコードを参照するための Foreign Function Interface (FFI) を持っている．これにより，VMDL は簡潔な言語仕様を持ちながら，C 言語上で定義された複雑な型に対して操作することができる．以降，VMDL 上で定義された関数を VMDL 関数，C 言語上で定義された関数を C 関数と呼ぶ.

3.2 構文

図 3.1 に VMDL の構文を示す.

VMDL のプログラム ⟨toplevel⟩ は，複数のユニオン型定義 ⟨union-definition⟩，externC 定義 ⟨externC-definition⟩，関数定義 ⟨function-definition⟩ から成る．ユニオン型定義 ⟨union-definition⟩ は，複数の VM 内部データ型をまとめた型を定義する．型システムについては 3.3 節で詳細を述べる．externC 定義 ⟨externC-definition⟩ を用いることで，VMDL 上から C 言語上で定義された型，関数，定数，変数を参照することができる．C 言語との FFI については 3.2.2 節で詳細を述べる．

関数定義 ⟨function-definition⟩ を用いて VMDL 関数を定義する．関数はアノテーション ⟨annotations⟩ を持つことができる．関数アノテーションについては 3.2.1 節で詳細を述べる．

```
v ∈ variable name              f ∈ function name         p ∈ pattern name
t ∈ type name in VMDL          l ∈ label

              ⟨toplevel⟩ ::= (⟨union-definition⟩ | ⟨externC-definition⟩ | ⟨function-definition⟩)∗
      ⟨union-definition⟩ ::= union t = t1 || t2 || ...
    ⟨externC-definition⟩ ::= ⟨ctype-definition⟩ | ⟨cfunction-definition⟩ | ⟨cconstant-definition⟩
                           | ⟨cvariable-definition⟩
      ⟨ctype-definition⟩ ::= externC type v (= ⟨ctype⟩)? | ⟨cmapping-definition⟩
   ⟨cmapping-definition⟩ ::= externC mapping heap? t {t1 v1 ; t2 v2 ; ...} ⟨ctype⟩?
  ⟨cfunction-definition⟩ ::= externC ⟨annotations⟩? function f : t
  ⟨cconstant-definition⟩ ::= externC constant v = ⟨cvalue⟩ : t
  ⟨cvariable-definition⟩ ::= externC variable v : t
   ⟨function-definition⟩ ::= ⟨annotations⟩? f : (td1 , td2 , ...) -> tr ⟨function-body⟩
         ⟨function-body⟩ ::= f (v1 , v2 , ...) ⟨block⟩
             ⟨statement⟩ ::= ⟨block⟩ | ⟨match⟩ | ⟨return⟩ | ⟨assignment⟩ | ⟨declaration⟩
                           | ⟨if⟩ | ⟨do⟩ | ⟨while⟩ | ⟨expression⟩;
                 ⟨block⟩ ::= {⟨statement⟩∗ }
                 ⟨match⟩ ::= (l :)? match (v1 , v2 , ...) {⟨case⟩∗ }
                  ⟨case⟩ ::= case ( ⟨pattern⟩ ) ⟨pattern-body⟩
               ⟨pattern⟩ ::= t v | !⟨pattern⟩ | (⟨pattern⟩)
                           | ⟨pattern⟩ && ⟨pattern⟩ | ⟨pattern⟩ || ⟨pattern⟩ | true
          ⟨pattern-body⟩ ::= {(⟨statement⟩ | ⟨rematch⟩)∗ }
               ⟨rematch⟩ ::= rematch l (v1 , v2 , ...);
                ⟨return⟩ ::= return ⟨expression⟩;
            ⟨assignment⟩ ::= v <- ⟨expression⟩;
           ⟨declaration⟩ ::= t v (= ⟨expression⟩)?;
                    ⟨if⟩ ::= if (⟨expression⟩) ⟨statement⟩ (else ⟨statement⟩)?
                    ⟨do⟩ ::= do ( int v = ⟨expression⟩ to ⟨expression⟩ step ⟨expression⟩ ) ⟨statement⟩
                 ⟨while⟩ ::= while ( ⟨expression ⟩ ) ⟨statement⟩
            ⟨expression⟩ ::= ⟨expression⟩ ? ⟨expression⟩ : ⟨expression⟩ | ⟨expression⟩ ⟨operators⟩ ⟨expression⟩
                           | ⟨function-call⟩ | ⟨unary-operators⟩⟨expression⟩ | ⟨primary-expression⟩
    ⟨primary-expression⟩ ::= ⟨constant⟩ | "⟨string⟩" | (⟨expression⟩) | v
             ⟨operators⟩ ::= || | && | | | ^ | & | == | != | <= | >= | < | > | << | >> | + | - | * | /
       ⟨unary-operators⟩ ::= + | - | ~ | !
         ⟨function-call⟩ ::= f (⟨expression⟩, ⟨expression⟩, ...)
              ⟨constant⟩ ::= constant literal of float or integer or character or boolean
                ⟨string⟩ ::= string value
           ⟨annotations⟩ ::= (⟨annotation⟩, ⟨annotation⟩, ...)
            ⟨annotation⟩ ::= vmInstruction | needContext | triggerGC | makeInline
                           | builtinFunction | calledFromC
                 ⟨ctype⟩ ::= A type name in c-language
                ⟨cvalue⟩ ::= direct c-language code in string
```

図 3.1　VMDL の構文

関数は本体としてブロック ⟨block⟩ を持つ．

文 ⟨statement⟩ は，複数の文を持つブロック ⟨block⟩，型ディスパッチ処理を行う `match` 文 ⟨match⟩，関数から制御を返す `return` 文 ⟨return⟩，変数へ値を代入する代入文 ⟨assignment⟩，変数を宣言する宣言文 ⟨declaration⟩，条件分岐処理を行う `if` 文 ⟨if⟩，反復処理を行う `do` 文 ⟨do⟩， `while` 文 ⟨while⟩，および式文がある．

型ディスパッチ処理を行う `match` 文 ⟨match⟩ は，VMDL が持つ不要な機能の削減のために重要な構文である．ラベル l を与えることができ，任意の多変数 v_1，v_2，... に対して型ディスパッチ処理をする．型ディスパッチ処理のそれぞれの分岐を `case` 節 ⟨case⟩ と呼び，`case` 節は自身の型条件を表すパタン ⟨pattern⟩ と `case` 節本体 ⟨pattern-body⟩ を持つ．パタン ⟨pattern⟩ は，変数 v が型 t であるという型条件を “$t\ v$” と表記し，否定，論理積，論理和も用いることができる．また，常に真である型条件を表す `true` を使うこともできる．`case` 節本体 ⟨pattern-body⟩ は，基本的にはブロックと同じであるが，`rematch` 文 ⟨rematch⟩ を記述できる．`rematch` 文 ⟨rematch⟩ は，指定したラベル l が与えられた `match` 文へ，変数を v_1，v_2，... に置き換えた上で再度型ディスパッチ処理をする．これにより，図 2.2 の 9 行目と 17 行目のように，vmgen では再ディスパッチ先にラベルを貼った上でそのラベルへ `goto` 文を用いてジャンプ処理を手動で記述していたものを，型解析によって自動生成することが可能となっている．

式 ⟨expression⟩ は，単項，二項，三項演算子に加えて関数呼び出し式を使用することができ，特に二項演算子に関しては C 言語データ型に対応する型に対する四則演算や比較演算，ビット演算などが用意されており，それら単純な計算ができるように設計されている．これらの演算で不十分な処理については，`externC` 関数の呼び出しを用いることになる．

3.2.1 関数のアノテーション

VMDL 関数，C 関数ともにアノテーション ⟨annotations⟩ を付与することができる．用意されているアノテーションとその意味を表 3.1 に示す．

表 3.1 関数のアノテーション

アノテーション	意味
`vmInstruction`	関数は VM 命令定義である．
`needContext`	関数は実行にコンテキスト情報を必要とする．
`triggerGC`	関数は実行中に GC を起こしうる．
`makeInline`	関数のインライン展開（後述）最適化の対象である．
`builtinFunction`	関数は組み込み関数である．
`calledFromC`	関数は VM の C 言語コード部分から呼び出される．

`vmInstruction`: コンパイラが関数を C 言語コード片へ変換する際に，関数の引数を入力オペランド，返り値を出力オペランドとして特別な形にするために使用される．逆に，VM 開発者はこのアノテーションを付与するだけで，VM 命令を通常の VMDL 関数として定義することができる．

`needContext`: 図 2.2 の 22 行目の `to_double` 関数のように，VM 内部の関数はプログラムの進行状況を保持するコンテキスト情報を必要とするものが多く存在するが，関数呼び出し式を記述する際にコンテキスト情報を毎回明示的に渡すのは手間がかかる．このアノテーションが付与された関数は C 言語コード上ではコンテキスト情報を受け取るように変換されるため，VM 開発者はコンテキスト情報の受け渡しに関するコードを記述する必要がなくなる．

`triggerGC`: 図 2.2 の 22 行目の `to_double` 関数のように，GC を起こしうる関数はその前後に GC ルートへの退避操作を行うコードを挿入する必要がある．このアノテーションが付与された関数は GC を起こしうる関数としてコンパイラに認識され，生存変数解析処理とあわせてコンパイラが GC に関する操作を自動的に挿入する．そのため，VM 開発者は関数が GC を起こしうるかにのみ注意すればよくなる．

`builtinFunction`: このアノテーションが付与された関数は組み込み関数として扱われ，引数に組み込み関数のレシーバオブジェクトと組み込み関数の実引数が渡される．また，渡された引数の数などの情報も暗黙的に利用できるようになる．また，コンパイラが関数を C 言語コード片へ変換する際に，返り値を VM の実装に合わせた形にするために使用される．そのため，VM 開発者は VM 内部の組み込み関数の実装を意識せずに定義することができる．

`calledFromC`: このアノテーションが付与された関数は，C 言語コード部分から呼ばれることを表す．C 言語上からの呼び出しに関しては，静的な型検査ができないため，関数が期待する VM 内部データ型が渡されていることを検査するコードを自動的に挿入する．例えば，引数に `Special` 型を期待する `special_to_string` 関数がこのアノテーションを持つ場合，コンパイラは C 言語コード上の関数本体の先頭に引数が `Special` 型であるか検査する `assert` マクロを挿入する．

3.2.2 C 言語とのインターフェース

前述の通り，VMDL はポインタに関する低レベル操作や使用頻度の低い操作などを持っていない．そのような操作は C 言語との FFI を用いて実現される．図 3.1 中の ⟨externC-definition⟩ が C 言語との FFI であり，これらを用いて C 言語上で定義された型，関数，定数，変数を参照することができる．

例として，オブジェクトのレイアウトを表す `Shape` 構造体のポインタである `Shape*` 型を挙げる．VMDL 上で，新しく文字列オブジェクトを生成するために `externC` 関数 `new_string_object` を使用したいとする．しかし，`new_string_object` 関数は引数に `Shape*` 型の値を受け取る必要がある．なお，ここで渡すべき値は `g_shape_String` という名前のグ

ローバル変数が保持している．このとき，VMDL は Shape* 型についての操作を持つ必要はなく，C 言語上で “Shape*” という名前のついた型の値を受け渡しできれば充分である．このような場合，まず externC 型宣言を用いて，“Shape*” という名前のついた型が存在することを以下のように宣言する．

```
externC type ShapePtr = "Shape*"
```

ここでは，VMDL 上での型の名前は “ShapePtr” とし，C 言語上では “Shape*” と表記するという情報を与えた．次に，externC 変数宣言を用いて，C 言語上に g_shape_String という名前の変数が存在することを以下のように宣言する．

```
externC variable g_shape_String : ShapePtr
```

これにより，VMDL 側から見て C 言語上に ShapePtr 型の変数 g_shape_String が存在するという情報を与えた．また，以下のように externC 関数を宣言する．

```
externC (needContext, triggerGC)
  new_string_object : (cstring, ShapePtr, JSValue) -> StringObject
```

これで，new_string_object 関数が第 2 引数に Shape* 型の値を受け取ることを Shape* 型の詳細な操作を持たずに定義できる．VMDL 上の new_string_object 関数呼び出しは以下のように記述できる．

```
StringObject obj = new_string_object("str", g_shape_String, v);
```

他にも，VMDL は C 言語上の文字列に対する詳細な操作を持っておらず，externC 関数を用いて操作を行う．

3.3 データ型

VMDL が持つデータ型は，一部の型を除き，C 言語データ型と VM 内部データ型それぞれに対応する型に分けることができる．

C 言語データ型に対応する型として，整数型に対応する int 型，浮動小数点数型に対応する double 型，文字列（char*型）に対応する cstring 型などが定義されている．3.2 節で述べたように，特に int 型や double 型などの数値を表す C 言語データ型に対しては，四則演算などの操作が用意されている．また，3.2.2 節で述べたように，externC 型宣言を用いて C 言語データ型を導入した場合，VMDL 上では CValue 型と呼ばれる，ブラックボックス化された C 言語上の値であることを意味する型として扱われる．CValue 型は，その型を持つ変数を宣言して代入する，関数の引数に渡す，比較演算操作だけが可能であり，VMDL 上では値を操作できない．

VM 内部データ型に対応する型として，表 2.1 に示した型を利用することができる．また，複

数の VM 内部データ型と union 型をまとめて新しい union 型を定義することができる．union 型は，まとめた VM 内部データ型のいずれかを実行時に取るデータ型を意味する．例えば，実行時に `Fixnum` 型あるいは `Flonum` 型を取りうる union 型 `Number` 型を以下のようにして定義できる．

```
union Number = Fixnum || Flonum
```

なお，union 型は C 言語データ型については定義することができない．また，組み込みの union 型として， VM 内部データ型すべてを表す `JSValue` 型が定義されている．以降，VM 内部データ型と union 型をまとめて JSValue 系の型と呼ぶ．3.4 節で形式的に示すが，C 言語データ型を持つ変数と JSValue 系の型を持つ変数は独立して型付けされる．そのため，コンパイラはそれらの変数について独立して型解析をする．

3.4 型付け規則

本節は式および文の型付け規則を示す．型付け規則は，図 3.2 を用いて表現する．式 e の型判断について，型環境 Γ において式 e の取りうる型の集合が $\boldsymbol{\tau}$ であることを

$$\Gamma \vdash e : \boldsymbol{\tau}$$

と表記する．また，文 s の型判断は，次の六つ組で表記する．

$$(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, s, \boldsymbol{\Gamma}', \Upsilon')$$

ここで，$\mathcal{M}$ はラベルから対応する `match` を取得するためのマップ，$\boldsymbol{\Gamma}$ と Υ は文 s 直前の型環境の集合および未初期化変数集合であり，$\boldsymbol{\Gamma}'$ と Υ' は直後の型環境の集合および未初期化変数集合を表す．また，型付け規則において使用する補助関数を図 3.3 に示す．

3.4.1 式の型付け規則

定数

$$\Gamma \vdash \texttt{true} : \{\text{int}\} \qquad (\textsc{PrimTrue})$$

$$\Gamma \vdash \texttt{false} : \{\text{int}\} \qquad (\textsc{PrimFalse})$$

$$\Gamma \vdash i : \{\text{int}\} \qquad (\textsc{PrimInt})$$

$$\Gamma \vdash d : \{\text{double}\} \qquad (\textsc{PrimDouble})$$

$$\Gamma \vdash str : \{\text{cstring}\} \qquad (\textsc{PrimString})$$

変数

型環境 Γ において，変数 v の型が τ であることを $\Gamma(v) = \tau$ とする．

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau} \qquad (\textsc{Variable})$$

単項演算子

$\ominus \in \{$+, -, ~, !$\}$ は単項演算子とする．

また，$\ominus \in$ +, - $\Rightarrow \boldsymbol{\tau} \subseteq \mathrm{int}, \mathrm{double}$，$\ominus \in$ ~, ! $\Rightarrow \boldsymbol{\tau} = \mathrm{int}$ とする．

$$\frac{\Gamma \vdash e : \boldsymbol{\tau}_1}{\Gamma \vdash \ominus e : \boldsymbol{\tau}_1} \qquad (\textsc{UnaryOp})$$

$i \in Integer$	::=	整数
$d \in Float$	::=	浮動小数点数
$str \in String$	::=	文字列
$v \in Variable$	::=	変数
$vs \in Variables$	::=	$(v_1, v_2, \ldots)$
$f \in Function$	::=	関数名
$e \in Expression$	::=	式
$s \in Statement$	::=	文
$p \in Pattern$	::=	`case` 節のパタン
$b \in Blockstatement$	::=	ブロック文
$l \in Label$	::=	`match` 文のラベル
$\tau^c \in Ctype$	::=	C 言語データ型
$\tau^v \in VM\ internal\ datatype$	::=	VM 内部データ型
$\tau^u \in Uniontype$	::=	union 型
$\tau \in type$	::=	$\tau^c \mid \tau^v \mid \tau^u$
$\boldsymbol{\tau} \in Nonempty\ set\ of\ types$	::=	$\{\tau_1, \tau_2, \ldots\}$
$\Gamma \in Type\ Environment$	::=	$\{v_1 \mapsto \tau_1, v_2 \mapsto \tau_2, \ldots\}$
$\boldsymbol{\Gamma} \in Nonempty\ set\ of\ \Gamma$	::=	型環境の空でない集合
$\Upsilon \in Set\ of\ unssigned\ VM\ variables$	::=	$\{v_1 \mapsto \boldsymbol{\tau}_1, v_2 \mapsto \boldsymbol{\tau}_2, \ldots\}$
$\mathcal{M} \in Match\ Map$	::=	$\{l_1 \mapsto \langle \boldsymbol{\Gamma}_1, vs_1\rangle, l_2 \mapsto \langle \boldsymbol{\Gamma}_2, vs_2\rangle, \ldots\}$

図 3.2　型付け規則の要素

二項演算子

$\oplus$ は二項演算子を表すものとする．演算の結果の型を求める補助関数 $binop$ は，直感的には演算子 $\oplus$ と被演算子 v_1，v_2 について C 言語の規則のような型付けを行う．

$$\frac{\Gamma \vdash e_1 : \boldsymbol{\tau}_1 \qquad \Gamma \vdash e_2 : \boldsymbol{\tau}_2 \\ \boldsymbol{\tau}_r = \{binop(\oplus, \tau_1, \tau_2) \mid \tau_1 \in \boldsymbol{\tau}_1,\ \tau_2 \in \boldsymbol{\tau}_2\}}{\Gamma \vdash e_1 \oplus e_2 : \boldsymbol{\tau}_r} \qquad (\textsc{BinaryOp})$$

三項演算子

$$\frac{\Gamma \vdash e_1 : \mathrm{int} \qquad \Gamma \vdash e_2 : \boldsymbol{\tau}_2 \qquad \Gamma \vdash e_1 : \boldsymbol{\tau}_3}{\Gamma \vdash e_1 \mathbin{?} e_2 : e_3 : \boldsymbol{\tau}_2 \cup \boldsymbol{\tau}_3} \qquad (\textsc{TernaryOp})$$

関数適用

$$\frac{\Gamma \vdash f : \{(\boldsymbol{\tau}_{p1}, \boldsymbol{\tau}_{p2}, \ldots) \to \boldsymbol{\tau}_r\} \qquad \forall i, \Gamma \vdash e_i : \boldsymbol{\tau}_i \qquad \forall i, \boldsymbol{\tau}_i \subseteq \boldsymbol{\tau}_{pi}}{\Gamma \vdash f(e_1, e_2, \ldots) : \boldsymbol{\tau}_r} \qquad (\textsc{FuncCall})$$

3.4.2 文の型付け規則

宣言文と代入文は，C 言語データ型の変数と JSValue 系の型の変数でルールが異なる．

式文

$$\frac{\forall \Gamma \in \boldsymbol{\Gamma}, \Gamma \vdash e : \boldsymbol{\tau}}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, e;, \boldsymbol{\Gamma}, \Upsilon)} \qquad (\textsc{ExprStmt})$$

宣言文

$$(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, \tau^c\ v;, \{[v \mapsto \tau^c]\Gamma \mid \Gamma \in \boldsymbol{\Gamma}\}, \Upsilon) \qquad (\textsc{CVarDeclStmt})$$

$$\frac{\forall \Gamma \in \boldsymbol{\Gamma}, \Gamma \vdash e : \tau^c}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, \tau^c\ v = e;, \{[v \mapsto \tau^c]\Gamma \mid \Gamma \in \boldsymbol{\Gamma}\}, \Upsilon)} \qquad (\textsc{CVarDeclStmt2})$$

$$(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, \tau^v\ v;, \boldsymbol{\Gamma}, [v \mapsto \{\tau^v\}]\Upsilon) \qquad (\textsc{JSVarDeclStmt})$$

$$\frac{\boldsymbol{\tau} = \text{set of VM internal datatypes in } \tau^u}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, \tau^u\ v;, \boldsymbol{\Gamma}, [v \mapsto \boldsymbol{\tau}]\Upsilon)} \qquad (\textsc{JSVarDeclStmt2})$$

$$\frac{\forall \Gamma \in \boldsymbol{\Gamma}, \Gamma \vdash e : \tau^v}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, \tau^v\ v\ \texttt{=}\ e\texttt{;}, \boldsymbol{\Gamma}, [v \mapsto \{\tau^v\}]\Upsilon)} \quad (\textsc{JSVarDeclStmt3})$$

$$\frac{\begin{array}{c}\forall \Gamma \in \boldsymbol{\Gamma}, \\ \Gamma \vdash e : \boldsymbol{\tau}_e \qquad \boldsymbol{\tau}_e \subseteq \boldsymbol{\tau} \\ \boldsymbol{\tau} = \text{set of VM internal datatypes in } \tau^u\end{array}}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, \tau^u\ v\ \texttt{=}\ e\texttt{;}, \boldsymbol{\Gamma}, [v \mapsto \boldsymbol{\tau}]\Upsilon)} \quad (\textsc{JSVarDeclStmt4})$$

代入文

$$\frac{\begin{array}{c}\forall \Gamma \in \boldsymbol{\Gamma}, \\ \tau^c = \Gamma(v) \qquad \Gamma \vdash e : \{\tau^c\}\end{array}}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, v\ \texttt{<-}\ e\texttt{;}, \boldsymbol{\Gamma}, \Upsilon)} \quad (\textsc{CVarAssignStmt})$$

$$\frac{\begin{array}{c}\forall \Gamma \in \boldsymbol{\Gamma}, \\ \boldsymbol{\tau} = \Upsilon(v) \qquad \Gamma \vdash e : \boldsymbol{\tau}_e \qquad \boldsymbol{\tau}_e \subseteq \boldsymbol{\tau} \\ \boldsymbol{\Gamma}' = \{[v \mapsto \tau^v]\Gamma \mid \Gamma \in \boldsymbol{\Gamma}, \Gamma \vdash e : \boldsymbol{\tau}_e, \tau^v \in \boldsymbol{\tau}_e\}\end{array}}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, v\ \texttt{<-}\ e\texttt{;}, \boldsymbol{\Gamma}', \mathit{remove}(\Upsilon, v))} \quad (\textsc{JSVarAssignStmt})$$

`if` 文

`if` 文において，各分岐の直後における未初期化変数集合は同一でなければならない．言い換えれば，`if` 文の直前で未初期化であった変数について，片方の経路でのみ初期化をしてはならない．

$$\frac{\begin{array}{c}\forall \Gamma \in \boldsymbol{\Gamma}, \\ \Gamma \vdash e : \{\text{int}\} \qquad \Upsilon'_1 = \Upsilon'_2 \\ (\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, s_1, \boldsymbol{\Gamma}'_1, \Upsilon'_1) \qquad (\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, s_2, \boldsymbol{\Gamma}'_2, \Upsilon'_2)\end{array}}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, \texttt{if(}e\texttt{)}\ s_1\ \texttt{else}\ s_2, \boldsymbol{\Gamma}'_1 \cup \boldsymbol{\Gamma}'_2, \Upsilon'_1)} \quad (\textsc{IfStmt})$$

`do` 文

$$\frac{\begin{array}{c}\forall \Gamma \in \boldsymbol{\Gamma}, \\ \Gamma \vdash e_1 : \{\text{int}\} \qquad \Gamma \vdash e_2 : \{\text{int}\} \qquad \Gamma \vdash e_3 : \{\text{int}\} \\ (\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, s, \boldsymbol{\Gamma}, \Upsilon')\end{array}}{(\boldsymbol{\Gamma}, \Upsilon, \mathcal{M}, \texttt{do(int}\ v\ \texttt{=}\ e_1\ \texttt{to}\ e_2\ \texttt{step}\ e_3\texttt{)}\ s, \boldsymbol{\Gamma}, \Upsilon')} \quad (\textsc{DoStmt})$$

`while` 文

$$\frac{\forall\Gamma \in \mathbf{\Gamma}, \quad \Gamma \vdash e : \{\text{int}\} \qquad (\mathbf{\Gamma}, \Upsilon, \mathcal{M}, s, \mathbf{\Gamma}, \Upsilon')}{(\mathbf{\Gamma}, \Upsilon, \mathcal{M}, \texttt{while(}e\texttt{)}\ s, \mathbf{\Gamma}, \Upsilon')} \quad (\textsc{WhileStmt})$$

ブロック文

$$\frac{\begin{array}{c}(\mathbf{\Gamma}, \Upsilon, \mathcal{M}, s_1, \mathbf{\Gamma}_1, \Upsilon_1) \\ (\mathbf{\Gamma}_{i-1}, \Upsilon_{i-1}, \mathcal{M}, s_i, \mathbf{\Gamma}_i, \Upsilon_i)(1 < i \leq n) \\ \mathbf{\Gamma}'_n = \{\mathit{remove}(\Gamma_n, \mathrm{dom}(\mathbf{\Gamma})) \mid \Gamma_n \in \mathbf{\Gamma}_n\}\end{array}}{(\mathbf{\Gamma}, \Upsilon, \mathcal{M}, \texttt{\{}s_1\ s_2\ \ldots\ s_n\texttt{\}}, \mathbf{\Gamma}'_n, \Upsilon_n)} \quad (\textsc{BlockStmt})$$

`match` 文

$$\frac{\begin{array}{c}\mathbf{\Gamma} \subseteq \mathbf{\Gamma}' \qquad \mathbf{\Gamma}'_i = \mathit{limit}(vs, (p_1, \ldots, p_{i-1}), p_i, \mathbf{\Gamma}') \\ (\mathbf{\Gamma}'_i, \Upsilon, [l \mapsto \langle \mathbf{\Gamma}', vs \rangle]\mathcal{M}, b_i, \mathbf{\Gamma}''_i, \Upsilon''_i) \\ \mathbf{\Gamma}' = \bigcup_i \mathbf{\Gamma}''_i \qquad \forall i. \Upsilon''_i = \Upsilon'\end{array}}{(\mathbf{\Gamma}, \Upsilon, \mathcal{M}, l\texttt{:match(}vs\texttt{)\{case(}p_1\texttt{)}b_1 \cdots \texttt{case(}p_n\texttt{)}b_n\texttt{\}}, \mathbf{\Gamma}', \Upsilon')} \quad (\textsc{MatchStmt})$$

`rematch` 文

ラベル l に紐付いた `match` 文のディスパッチ対象の変数 v_1，...，v_n について，`reamtch` 文に与えられた v'_1，...，v'_n の型環境 Γ における型で置き換える．この置き換えた型環境を `match` 文の先頭における型環境 $\mathbf{\Gamma}_l$ へ追加する．`rematch` 文の後，`match` 文の先頭へ制御が移るため，直後の型環境は空集合となる．

$$\frac{\forall\Gamma \in \mathbf{\Gamma}, \quad \mathcal{M}(l) = \langle \mathbf{\Gamma}_l, (v_1, \ldots, v_m) \rangle \qquad [v_1 \mapsto \Gamma(v'_1)] \cdots [v_m \mapsto \Gamma(v'_m)]\Gamma \in \mathbf{\Gamma}_l}{(\mathbf{\Gamma}, \Upsilon, \mathcal{M}, \texttt{rematch}\ l\ (v'_1, \ldots, v'_m)\texttt{;}, \{\}, \Upsilon)} \quad (\textsc{RematchStmt})$$

3.5 VMDL 関数の記述例

本節は，VM の命令定義，型変換関数，組み込み関数の VMDL による記述例を挙げる．

図 3.4 に，簡略化した `add` 命令の VMDL による定義を示す．1 行目と 2 行目は JavaScript における数値型に対応する `Number` 型と，型ディスパッチ処理の型条件件に使用する `ffs` 型を定義している．4 行目から 9 行目は，C 言語上に実装された各種関数についての `externC` 関数宣言である．`to_double` 関数などは，GC を引き起こしうる関数であるため，`triggerGC` アノテーションが付与されている．11 行目からが `add` 命令の定義（以降，単に `add` 関数と呼ぶ.）

の本体である．add 関数は vmInstruction アノテーションが付与されており，これによってコンパイラはこの関数を VM 命令の形でコンパイルする．add 関数は JSValue 型の変数を 2 つ受け取り，JSValue 型の値を返す．これは add 命令が入力オペランドを 2 つ取り，デスティネーションを持つことに対応する．add 関数本体は 14 行目から始まる top とラベル付けされた match 文で構成されており，変数 v1，v2 の型の組み合わせに対して型ディスパッチ処理をする．分岐は 16 行目，21 行目，25 行目，31 行目のそれぞれの case 節で定義されていて，各節は型の条件を持っている．処理の内容は基本的に図 2.2 に示した vmgen による定義と同じであるが，いくつかの点で異なっている．まず，18 行目で型変換に使用している関数が to_int に

binop (op, v_1, v_2) {
  if *op* is a comparison operator then
    return { int }
  else
    return {the higher data type of the v1 and v2}
}

typevecs (vs, P) {
  return $\{\boldsymbol{\tau} \mid vs : \boldsymbol{\tau}$ matches $P\}$
}

limit $(vs, \langle P_0, \ldots, P_{i-1}\rangle, P_i, \boldsymbol{\Gamma})$ {
  $\boldsymbol{\tau} = typevecs(vs, P_i) - \bigcup_{j=0}^{i-1} typevecs(vs, P_j)$
  return $\bigcup_{k,l}\{update(\Gamma_l, vs, \boldsymbol{\tau}_k) \mid \boldsymbol{\tau}_k \in \boldsymbol{\tau}, \Gamma_l \in \boldsymbol{\Gamma}\}$
}

update $(\Gamma, vs, \boldsymbol{\tau})$ {
  for $(\langle v, \tau\rangle \in \text{zip}(vs, \boldsymbol{\tau}))$
    $\Gamma = [v \to \tau]\Gamma$
  return Γ
}
remove $(\Upsilon, v) = \{u \mapsto \boldsymbol{\tau} | u \mapsto \boldsymbol{\tau} \in \Upsilon, u \neq v\}$

図 3.3　型付け規則の補助関数

```
 1  union Number = Fixnum || Flonum
 2  union ffs = Fixnum || Flonum || Special
 3
 4  externC function to_int : JSValue -> int
 5  externC (needContext, triggerGC) function int_to_number : int -> Number
 6  externC (needContext, triggerGC) function ejs_string_concat : (String, String) -> String
 7  externC (needContext, makeInline, calledFromC) function to_string : JSValue -> String
 8  externC (needContext, triggerGC) function to_double : JSValue -> double
 9  externC (needContext, triggerGC) function double_to_number : double -> Number
10
11  (vmInstruction, needContext, triggerGC) add : (JSValue, JSValue) -> JSValue
12  add (v1, v2) {
13    //(1)
14    top: match (v1, v2) {
15      //(2)
16      case (Fixnum v1 && Fixnum v2) {
17        //(3)
18        int s = to_int(v1) + to_int(v2);
19        return int_to_number(s);
20      }
21      case (String v1 && String v2) {
22        //(4)
23        return ejs_string_concat(v1, v2);
24      }
25      case (String v1 && ffs v2) {
26        //(5)
27        String s = to_string(v2);
28        //(6)
29        rematch top(v1, s);
30      }
31      case (true) {
32        //(7)
33        double u1 = to_double(v1);
34        double u2 = to_double(v2);
35        return double_to_number(u1 + u2);
36      }
37    }
38  }
```

図 3.4　add 命令定義（簡略版）

なっている．ここで，VMDL における int 型は C 言語の cint 型に対応していることに注意されたい．vmgen 上の C 言語による実装とは異なり，VM 開発者は型変換の引数の型に関して最適化して記述する必要はない．C 言語を用いる場合，このようなプログラムは冗長な型ディスパッチ処理を引き起こし，プログラムの効率を悪化させてしまう．しかし，VMDL を用いる場合，18 行目の位置において to_int 関数の引数 v1 が Fixnum 型であることを解析することができる．そのため，コンパイラによる後述の最適化処理によって，冗長な型ディスパッチ処理を回避する．次に，vmgen による実装では goto 文を使用していた，21 行目の case 節とそこへジャンプしていた 29 行目の処理が，rematch 文を用いた記述に置き換わっている．これによって，複数の分岐間をまたがる処理を型解析可能な形で記述できている．

```
1   union ffss = Fixnum || Flonum || String || Special
2
3   externC (makeInline, calledFromC) function fixnum_to_string : Fixnum-> String
4   externC (makeInline, calledFromC) function flonum_to_string : Flonum -> String
5   externC (makeInline, calledFromC) function special_to_string : Special-> String
6
7   (needContext, makeInline, calledFromC) to_string : ffss -> String
8   to_string(v){
9       match(v){
10          case(String v){
11              return v;
12          }
13          case(Fixnum v){
14              return fixnum_to_string(v);
15          }
16          case(Flonum v){
17              return flonum_to_string(v);
18          }
19          case(Special v){
20              return special_to_string(v);
21          }
22      }
23  }
```

図 3.5　to_string 関数の定義（簡略版）

図 3.5 に，簡略化した to_string 関数の VMDL による定義を示す．この関数は，図 3.4 の add 関数でも使用されている，VM 内部で使用される型変換関数である．関数の定義は単純で，引数 v に対して 7 行目から始まる型ディスパッチ処理を行い，データ型に応じて値を返却する．例えば，v が Fixnum 型のときには，fixnum_to_string(v) の値を返却する．

図 3.6 に，簡略化した array_every 関数の VMDL による定義を示す．array_every 関数は，ECMAScript の言語仕様 [6] において定義された，Array のプロトタイプオブジェクトが持つ組み込み関数のひとつである every メソッド[*1]の実装である．every 関数は，レシーバオブジェクトの各要素すべてが与えられたコールバック関数を満たすか否かを返す．array_every 関数は builtinFunction アノテーションが付与されており，これによってコンパイラはこの関数を組み込み関数としてコンパイルする．このアノテーションを持つ関数内では，every 関数に与えられた実引数の数を na によって参照できる．array_every 関数の引数は 3 つで，第一引数 a にはレシーバオブジェクトが，第二引数以降に組み込み関数に渡された引数が渡される．第二引数に与えられるコールバック関数は，VM 内部データ型の Function 型または Builtin 型であることが求められるため，2 行目でそのような union 型を Callable として定義している．21 行目から 45 行目までが関数の本体で，while 文を用いて，レシーバオブジェクトの各要素について順番に関数 fn を適用させている．関数 fn は Callable 型であり，Function 型と Builtin 型で関数の呼び出し処理が異なるため，30 行目の match 文で型ディスパッチ処理をし

[*1] https://262.ecma-international.org/5.1/#sec-15.4.4.16

```
 1  union Number = Fixnum || Flonum
 2  union Callable = Function || Builtin
 3
 4  externC constant JS_TRUE = "JS_TRUE" : Special
 5  externC constant JS_FALSE = "JS_FALSE" : Special
 6  externC constant JS_UNDEFINED = "JS_UNDEFINED" : Special
 7  externC function to_boolean : JSValue -> Special
 8  externC function number_to_int : Number -> int
 9  externC (needContext, triggerGC) function int_to_number : int -> Number
10  externC function get_jsarray_length : Array -> Number
11  externC (triggerGC) function get_array_element : (Array, int) -> JSValue
12  externC (triggerGC) function set_array_element : (Array, int, JSValue) -> void
13  externC function has_array_element: (JSValue, int) -> int
14  externC (triggerGC)
15    function send_function3 : (JSValue,Function,JSValue,JSValue,JSValue) -> JSValue
16  externC (triggerGC)
17    function send_builtin3 : (JSValue,Builtin,JSValue,JSValue,JSValue) -> JSValue
18
19  (builtinFunction, needContext, triggerGC)
20  array_every : (Array, Callable, JSValue) -> JSValue
21  array_every (a, fn, this) {
22    int len = number_to_int(get_jsarray_length(o));
23    JSValue t = na >= 2 ? this : JS_UNDEFINED;
24    int k = 0;
25    while (k < len) {
26      if (has_array_element(a, k)) {
27        Number jsk = int_to_number(k);
28        JSValue val = get_array_element(a, k);
29        JSValue result;
30        match (fn) {
31          case (Function fn) {
32            result <- send_function3(t, fn, val, jsk, a);
33          }
34          case (Builtin fn) {
35            result <- send_builtin3(t, fn, val, jsk, a);
36          }
37        }
38        if (result == JS_FALSE || (result != JS_TRUE && to_boolean(result) == JS_FALSE)) {
39          return JS_FALSE;
40        }
41      }
42      k <- k + 1;
43    }
44    return JS_TRUE;
45  }
```

図 3.6　array_every 関数の定義（簡略版）

ている．

4 VMDLC

4.1 概要

本研究では，VMDL のコンパイラである VMDLC (VMDL Compiler) を実装した．VMDLC は，VMDL で記述されたソースコードを受け取り，それを C 言語のコード片へ変換する．なお，変換した C 言語コードは C コンパイラによってコンパイルされるため，一般的なコンパイラが行う各種最適化を VMDLC は行わない．代わりに，C 言語レベルでは最適化のできない，VM 内部データ型に関する最適化処理をいくつか実装している．この章では，記述例を用いた VMDLC による静的な型解析処理の流れと，解析結果を用いた型ディスパッチ処理削減の方法，アノテーションの処理，および VM 内部データ型に関する最適化処理について述べる．

4.2 型解析処理

VMDLC は，プログラムの各点における型環境，変数とその変数が取りうるデータ型の対応関係を計算する．基本的に，解析はプログラムの先頭から末尾に向かって進められる．型環境は，「変数名対データ型」の集合の集合として表現され，`JSValue` 系のデータ型は VM 内部データ型単位で保持される．例えば，プログラムのある地点において，変数 `v` が定義されており，`Fixnum` 型と `Flonum` 型の union 型である `Number` 型の値を持つとする．このとき，型環境は $\{\{\mathtt{v} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v} \mapsto \mathtt{Flonum}\}\}$ と表現する．これにより，複数の変数が同じデータ型しか取らないことを表現する．例えば，この後に「`Number w = v;`」のような，変数 `w` を `v` の値で初期化する宣言文があったとする．この宣言文直後の型環境は，$\{\{\mathtt{v} \mapsto \mathtt{Fixnum}, \mathtt{w} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v} \mapsto \mathtt{Flonum}, \mathtt{w} \mapsto \mathtt{Flonum}\}\}$ となる．この型環境は，union 型を用いた $\{\mathtt{v} \mapsto \mathtt{Number}, \mathtt{w} \mapsto \mathtt{Number}\}$ という表現方法とは異なり，変数 `v` と `w` の関係を厳密に表現できる．以降，「変数名対データ型」の集合の集合を単に型環境と呼ぶ．

型解析処理は，スペックで与えられた情報を基に関数の入り口における型環境が与えられ，プログラムの制御フローに沿って解析が進められる．3.4 で示したとおり，`if` 文のような分岐の末尾において，型環境は和集合を取って結合される．また，`match` 文先頭の型環境は `rematch` 文による合流が発生する可能性があるため，変化しなくなるまで繰り返し型環境を拡張する．

図 3.4 に示した `add` 関数を例に型解析処理の流れを説明する．この `add` 関数は `rematch` 文を持つため，型環境を 2 周計算する．この例において，以下の命令スペックが与えられたとする．

```
add (-,Fixnum,Fixnum) accept
add (-,String,Fixnum) accept
```

```
add (-,_,_) unspecified
```

これにより，位置 (1) における型環境は，

$$\{\{\mathtt{v1} \mapsto \mathtt{Fixnum}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}\}$$

となる．1 周目の型解析では関数本体の先頭に来ている match 文の入り口，位置 (2) の型環境も同じである．次に，match 文による型ディスパッチ処理の内部では，位置 (2) の型環境より，1 番目と 3 番目の case 節が選択される．1 番目の case 節内部の先頭，位置 (3) の型環境は，

$$\{\{\mathtt{v1} \mapsto \mathtt{Fixnum}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}\}$$

であり，3 番目の case 節内部の先頭，位置 (5) の型環境は，

$$\{\{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}\}$$

となる．残りの case 節，2 番目と 4 番目の case 節内部の先頭，位置 (4) と 位置 (7) の型環境は，どちらも空集合となる．

3 番目の case 節内部では，ローカル変数 s が宣言され，String 型の値で初期化されている．そのため，直後の位置 (6) における型環境は，

$$\{\{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}, \mathtt{s} \mapsto \mathtt{String}\}\}$$

となる．直後の 29 行目の rematch 文により，v1 に v1 を，v2 に s を代入して，top とラベル付けされた match 文で再び型ディスパッチ処理をする．ここまでが 1 周目の型解析処理である．表 4.1 の 1 周目の列に各点における型環境を示す．

29 行目の rematch 文により，位置 (2) の型環境が拡張され，

$$\{\{\mathtt{v1} \mapsto \mathtt{Fixnum}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{String}\}\}$$

となる．match 文の先頭における型環境が変化したため，2 周目の型解析処理が発生する．2 周目の型解析処理では，位置 (2) の型環境より，1 番目，2 番目と 3 番目の case 節が選択される．2 番目の case 節内部の先頭，位置 (4) の型環境は，

$$\{\{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{String}\}\}$$

となる．2 番目の case 節は rematch 文を持たず，また，1 番目と 3 番目の case 節内部の先頭の位置における型環境は変化しないため，これ以上 match 文先頭の位置 (2) の型環境は変化しないため，この match 文に関する型解析処理は終了し，add 関数はこれ以降に文を持たないため，全体の型解析処理も終了となる．表 4.1 の 2 周目の列に最終的な各点における型環境を示す．型解析処理の結果，4 番目の case 節内部の先頭の位置 (7) における型環境が空集合となったことがわかる．このような case 節は，実行されないことを意味している．そのため，VMDLC はそのような型環境が空集合の case 節を生成するコードから削減することで，型ディスパッチ処理に関してスペックに基づいた最小限の機能を持つコードを生成する．

表 4.1　図 3.4 の各点における型環境

	1 周目	2 周目
(1)	$\{\{\mathtt{v1} \mapsto \mathtt{Fixnum}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}\}$	（左に同じ）
(2)	$\{\{\mathtt{v1} \mapsto \mathtt{Fixnum}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}\}$	$\{\{\mathtt{v1} \mapsto \mathtt{Fixnum}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}, \{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{String}\}\}$
(3)	$\{\{\mathtt{v1} \mapsto \mathtt{Fixnum}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}\}$	（左に同じ）
(4)	$\{\}$	$\{\{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{String}\}\}$
(5)	$\{\{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}\}\}$	（左に同じ）
(6)	$\{\{\mathtt{v1} \mapsto \mathtt{String}, \mathtt{v2} \mapsto \mathtt{Fixnum}, \mathtt{s} \mapsto \mathtt{String}\}\}$	（左に同じ）
(7)	$\{\}$	（左に同じ）

4.2.1　変数のデータ型

3.3 節で述べたとおり，union 型として宣言された JSValue 系の型を持つ変数は union 型を構成する任意の VM データ型の値を代入することができる．型解析処理においては，型環境におけるそのような変数のデータ型は，宣言された型ではなく，実際に割り当てられた値の型となる．例えば，以下のような VMDL コードを考える．

```
f(JSValue x) {
  JSValue v;
  if (...) {
    v <- to_string(x); // (1)
  } else {
    v <- to_flonum(x); // (2)
  }
  // (3)
  ...
}
```

変数 v は JSValue 型と宣言されているが，位置（1）で String 型の値が，位置（2）で Flonum 型の値が代入されている．位置（1）の直後の型環境は，

$$\{\{\mathtt{x} \mapsto \mathtt{JSValue}, \mathtt{v} \mapsto \mathtt{String}\}\}$$

であり，位置（2）の直後の型環境は，

$$\{\{\mathtt{x} \mapsto \mathtt{JSValue}, \mathtt{v} \mapsto \mathtt{Flonum}\}\}$$

となる．この結果，位置（3）の型環境は，これらの和集合となり，

$$\{\{\mathtt{x} \mapsto \mathtt{JSValue}, \mathtt{v} \mapsto \mathtt{String}\}, \{\mathtt{x} \mapsto \mathtt{JSValue}, \mathtt{v} \mapsto \mathtt{Flonum}\}\}$$

となる．

4.3 型変換関数スペックの生成

型変換関数は，主に VM 命令や組み込み関数から呼び出される．そのため，型変換関数の引数に与えられるデータ型は，VM 命令と組み込み関数のスペックによって決定する．これらのスペックの情報を利用することによって，アプリケーションの開発者が VM 内部で使用される型変換関数のスペックを作ることなく，自動的に得ることができる．VMDLC は，VM 命令や組み込み関数をコンパイルする際に，型解析処理の結果を用いて型変換関数スペックを生成する機構を備えている．これにより，フレームワークは VMDL 関数間の型解析を行い，最小限の VM コードを生成する．

ここでは，簡単な例として，図 3.4 に示した add 命令と，図 3.5 に示した to_string 関数を挙げる．まず，add 命令に以下の命令スペックを与えたとする．

```
add (-,String,Fixnum) accept
add (-,String,Flonum) accept
add (-,_,_) unspecified
```

add 関数の 27 行目において to_string 関数が呼び出され，引数に与えられる変数 v2 のこの位置における型環境における型は Fixnum 型あるいは Flonum 型となる．そのため，VMDLC は to_string 関数に関して，以下の型変換関数スペックを生成する．

```
to_string (Fixnum) accept
to_string (Flonum) accept
to_string (_) unspecified
```

このように，すべての型変換関数に関して，必要な引数のデータ型の情報が書き出される．ただし，型変換関数の中には，C 言語コード部から呼び出されるものがある．そのような関数につい

では，VMDLC は必要な引数のデータ型の計算を行わず，「すべてのデータ型を受け付ける」スペックを生成する．そのような関数には，VM 開発者が calledFromC アノテーションを与えることで区別する．

4.4 関数アノテーションの処理

3.2.1 節で述べたとおり，関数に付与されるアノテーションには，コンパイル時に処理を挿入させるものがある．

まず，needContext アノテーションが付与された関数の呼び出し式は，第一引数にコンテキスト情報 context が渡される．例えば，VMDL 上で to_double(v); と記述された式は，C コード上では to_double(context, v); となる．また，VMDL 関数定義にアノテーションが以下のように付与されている場合，

```
(needContext) f : JSValue -> JSValue
f (v) {
  ...
}
```

C コード上では第一引数にコンテキスト情報 context が追加される．

```
JSValue f (Context *context, JSValue v) {
  ...
}
```

VMDL 上では，needContext アノテーションが付与された関数の呼び出し式を持つ関数定義にも needContext アノテーションを付与するように注意するだけでよい．

vmInstruction アノテーションが付与された VMDL 関数は，命令定義としてコンパイルされる．eJSVM において命令定義はスレッデッドコード [2] という手法を用いているため，コンパイラはその形式に沿ったコードを生成する．

builtinFunction アノテーションが付与された VMDL 関数は，組み込み関数としてコンパイルされる．eJSVM において組み込み関数は，コンテキスト情報 context，VM スタックの位置を表す fp，JavaScript 上で呼び出された時に与えられた引数の数 na を引数に受け取る関数として定義される．コンテキスト情報と fp を用いて， JavaScript 上で呼び出された時に与えられた引数を取得したり，VMDL 上で与えられた引数の名前と対応させたりする処理が自動的に挿入される．これにより，VM 開発者はそういった処理を記述せずに済む．例えば，図 3.6 で示した array_every 関数は，以下のようにコンパイルされる．

```
void array_every (Context* context, cint fp, cint na) {
```

```
  Value *args = (Value *)(&(get_stack(context, fp)));
  Value a = args[0];
  Value fn = args[1];
  Value this = args[2];
  ...
}
```

triggerGC アノテーションが付与された関数の呼び出し式は，その前後に GC ルートへの退避処理が挿入される．例えば，図 3.4 において，33 行目の to_double 関数はアノテーションが付与されているため，その後使用する変数 v2 の GC ルートへの退避処理が 33 行目前後に挿入される．

4.4.1 GC ルート退避処理の挿入アルゴリズム

eJSVM において，GC ルート退避処理の対象となるのは，VM ヒープ上に確保されるデータである．そのため，VMDL 上においては，VM ヒープ上のデータを指す VM 内部データ型が退避処理の対象となる．ただし，JSValue 系の型を持つ変数であっても，Fixnum 型と Special 型に関しては，即値を持つため退避処理の対象ではない．また，C 言語データ型の変数も対象ではない．GC ルートに登録された変数は，有効なアドレスを持つことが期待されているため，未初期化の変数は push してはならない．反対に，その後使用しない初期化済み変数が登録されていたとしても，関数が終了するまでに pop すれば問題はない．アルゴリズムは以上の点に注意する必要がある．

GC ルート退避処理の挿入は，制御フローグラフを用いた生存変数解析に基づいて行う．ここでは，図 4.1 に示す単純な関数 f を用いてアルゴリズムを説明する．ただし，関数内の「...v...」は変数 v を使う文とする．制御フローグラフは，基本ブロックと呼ばれる単位で構築する．基本ブロックとは，プログラムにおいて，途中で分岐や合流のないプログラムの列である．GC ルート退避処理の挿入は，基本ブロック単位で行う．

図 4.2 に，関数 f の制御フローグラフを示す．制御フローグラフは，関数の入口を表す ENTER ノードと 出口を表す EXIT ノードを持ち，それぞれのノードは制御の移る可能性のあるノードへの辺を持つ．ここでノードは基本ブロックに対応する．ノード N に関して，制御の移る可能性のあるノードの集合を $next(N)$，自身に移る可能性のあるノードの集合を $prev(N)$ と表記することとする．ここでは，関数の if 文の手前までの基本ブロックを表すノードを A ノード，if 文の then 部の基本ブロックを表すノードを B ノード，else 部の基本ブロックを表すノードを C ノードと呼ぶこととし，説明のため各ノードの基本ブロック中の文の列に番号を与えた．

変数が生存しているとは，ある文の地点において変数が有効な値を持っており，かつ，その文よりも後方でその変数が使用される可能性があることを指す．生存変数解析では，ノードが持つ

```
1     externC (needContext, triggerGC) function to_string: JSValue -> String
2     externC (needContext, triggerGC) function to_flonum: JSValue -> Flonum
3     externC function to_fixnum: JSValue -> Fixnum
4     externC (triggerGC) function try_gc : void -> void
5
6     (needContext, triggerGC) f : JSValue -> void
7     f (v) {
8       JSValue x = to_string(v);
9       JSValue y = to_flonum(v);
10      JSValue z = to_fixnum(v);
11      try_gc();
12      ...v...;
13      try_gc();
14      if(...){
15        try_gc();
16        ...x...y...;
17      }else{
18        try_gc();
19        ...y...z...;
20      }
21      return;
22    }
```

図 4.1　GC ルート退避処理が必要な関数の例

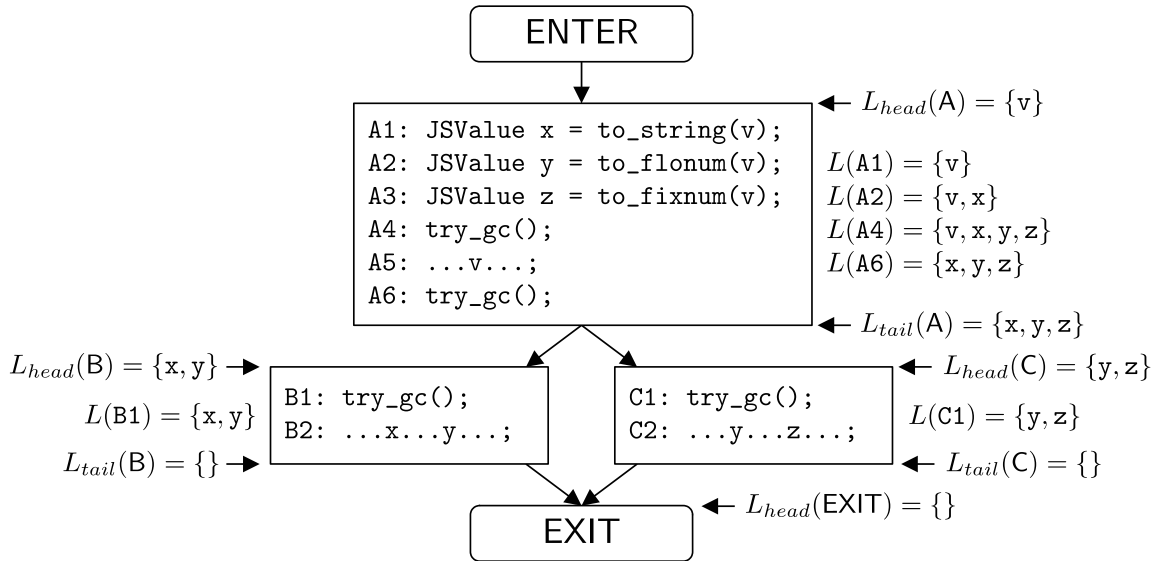


図 4.2　関数 `f` の制御フローグラフ

各文の位置における生存変数の集合を計算する．そのために，まず各ノード N の入口における生存変数集合を $L_{head}(N)$，出口における生存変数集合を $L_{tail}(N)$ と表記することとし，これらを計算する．$L_{tail}(N)$ は，$next(N)$ に含まれる各ノード（M_i とする）の $L_{head}(M_i)$ の和集合とすればよい．

$$L_{tail}(N) = \bigcup_i L_{head}(M)\ (M \in next(N))$$

$L_{tail}(N_i)$ を計算するために，各ノードの $L_{head}(N_i)$ を計算すればよいことになる．まず，EXIT ノードの入口における生存変数は存在しない．

$$L_{head}(\mathsf{EXIT}) = \{\}$$

その他のノードは，ノードが持つ文に関して，文の直後の生存変数集合から直前の生存変数集合を計算する．そのため，各ノード N は $L_{tail}(N)$ から始めて，文の列を末尾から先頭に向けて計算を進めることで $L_{head}(N)$ を求める．

ノードが持つ文は式文，代入文，宣言文の 3 つである，これらに関して直後の生存変数集合から直前の生存変数集合を計算する規則を示す．まず，式 e 中で使用している変数の集合を use という述語を用いて $use(e)$ と表記し，直後の生存変数集合 L に対して，文 s によって直前の生存変数集合が L' となることを (L, s, L') と表記することとする．

式文： 式 e で使用された変数を生存変数集合に追加する．

$$(L, e\texttt{;}, L \cup use(e))$$

代入文： 式文と同様に，式 e で使用された変数を生存変数集合に追加するが，代入される変数はこれ以前は未初期化の状態になるため，生存変数集合から除外する．

$$(L, v \texttt{ <- } e\texttt{;}, L \cup use(e) - \{v\})$$

宣言文： 式文と同様に，式 e で使用された変数が生存変数集合に追加される．宣言する変数はこれ以前は存在しないため，生存変数集合から除外される．

$$(L, \tau^u\ v \texttt{ = } e\texttt{;}, L \cup use(e) - \{v\})$$

これらの規則を用いて，ノード N が持つ文の列 s_1，...，s_n に対して，s_n から順番に直前の生存変数集合を計算することによって $L_{head}(N)$ を求める．以降，位置 i の直前における生存変数集合を $L(i)$ と表記する．規則を用いて，関数 `f` の制御フローグラフについて考える．まず，$L_{head}(\mathsf{EXIT}) = \{\}$ であり，ノード EXIT への遷移を持つノード B の $L_{tail}(B)$ と C の $L_{tail}(C)$ も空集合となる．ノード B に関して，`B2` で変数 `x` と `y` を使用しているため，$L(\texttt{B2}) = \{\texttt{x}, \texttt{y}\}$ となり，`B1` も同じである．これにより，$L_{head}(\mathsf{B}) = \{\texttt{x}, \texttt{y}\}$ となる．ノード C に関して，`C2` で変数 `y` と `z` を使用しているため，$L(\texttt{B2}) = \{\texttt{y}, \texttt{z}\}$ となり，`C1` も同じである．これにより，$L_{head}(\mathsf{C}) = \{\texttt{y}, \texttt{z}\}$ となる．ノード B，C の入口における生存変数集合が求まったため，ノード A の $L_{tail}(\mathsf{A})$ は，

$$\begin{aligned} L_{tail}(\mathsf{A}) &= L_{head}(\mathsf{B}) \cup L_{head}(\mathsf{C}) \\ &= \{\texttt{x}, \texttt{y}\} \cup \{\texttt{y}, \texttt{x}\} \\ &= \{\texttt{x}, \texttt{y}, \texttt{z}\} \end{aligned}$$

となる．そのため，ノード A が持つ最後の文の位置における生存変数集合は $L(\texttt{A6}) = \{\texttt{x}, \texttt{y}, \texttt{z}\}$ となる．`A5` で変数 `v` を使用しているため，$L(\texttt{A5}) = \{\texttt{v}, \texttt{x}, \texttt{y}, \texttt{z}\}$ となり，`A4` も同じである．`A3`

では変数 z を，A2 では y，A1 では x を宣言しているため，それぞれの位置における生存変数集合は $L(\mathtt{A3}) = \{\mathrm{v}, \mathrm{x}, \mathrm{y}\}$，$L(A2) = \{\mathrm{v}, \mathrm{x}\}$，$L(A1) = \{\mathrm{v}\}$ となる．これにより，$L_{head}(\mathsf{A}) = \{\mathrm{v}\}$ となる．以上で，すべてのノードとノードが持つ文に関して生存変数集合を求めることができた．

生存変数解析によって求めた生存変数集合を用いて，GC ルート退避処理を挿入する．GC ルートへの退避は，triggerGC アノテーションを持つ関数の呼び出しの前に VM が持つ GC 用のスタック領域へ push し，呼び出し後に pop する処理である．これを各基本ブロック単位で計算する．eJSVM 上では，変数 v に対する push を「GC_PUSH(v);」，pop を「GC_POP(v);」と表記する．また，複数の変数を一度に処理するマクロも定義されている．アルゴリズムとしては，一度退避した変数が関数呼び出し後に継続して退避しなければならない場合，その場では pop せず遅延させることで，冗長な push / pop を削減する．また，変数は GC ルートに登録する必要がなくなった段階で pop すべきであるが，スタックに積まれている順番によってはその場で pop することができない場合がある．そのようなケースを減らすために，スタックへ push する順番は，

1. 基本ブロックの先頭から末尾まで生存している変数
2. 基本ブロックの先頭では生存しているが途中で不要となる変数
3. 基本ブロック中で宣言された変数

とする．GC 用のスタック領域 G を用意し，プログラムの先頭から順番にスタックを計算して GC_PUSH/GC_POP を挿入する．

図 4.2 の制御フローグラフに，triggerGC アノテーションを持つ関数の呼び出し式を含む文の位置の生存変数集合を示す．まず，ノード A について考える．すべてのノードの先頭における G は空である．GC_PUSH/GC_POP を挿入すべき位置は，A1，A2，A4，A6 である．まず，A1 の位置で，G に積まれていない生存変数は v であるため，この直前に「GC_PUSH(v);」を挿入する．これにより，$G = [\mathrm{v}]$ となる．同様にして，A2 の位置で積まれていない生存変数を G に積む．A2 の 直前に「GC_PUSH(x);」を挿入し，$G = [\mathrm{v}, \mathrm{x}]$ となる．A4 の位置では，生存変数に y と z が加わるが，型環境において z は Fixnum 型をとることがわかっているため，処理の対象ではない．そのため，A4 の 直前に「GC_PUSH(y);」を挿入し，$G = [\mathrm{v}, \mathrm{x}, \mathrm{y}]$ となる．A6 では 生存変数がすべて G に積まれているため，特に処理は行わない．A6 以降に triggerGC アノテーションを持つ関数の呼び出し式を含む文は存在しないため，直後に「GC_PUSH3(y,x,v);」を挿入し，すべての変数を pop する．ここで G は空となる．

次に，ノード B について考える．GC_PUSH/GC_POP を挿入すべき位置は B1 である．この位置で生存変数は x と y が存在し，どちらも G に積まれていないため，この直前に「GC_PUSH2(x,y);」を挿入する．また，この位置以降に triggerGC アノテーションを持つ関数の呼び出し式を含む文は存在しないため，直後に「GC_POP2(y,x);」を挿入し，すべての変数を pop する．

```
1    void f (Context* context, JSValue v) {
2      GC_PUSH(v);
3      JSValue x = to_string(context, v);
4      GC_PUSH(x);
5      JSValue y = to_flonum(context, v);
6      JSValue z = to_fixnum(v);
7      GC_PUSH(y);
8      try_gc();
9      ...v...;
10     try_gc();
11     GC_POP3(y, x, v);
12     if (...) {
13       GC_PUSH2(y, x);
14       try_gc();
15       GC_POP2(x, y);
16       ...x...y...;
17     } else{
18       GC_PUSH(y);
19       try_gc();
20       GC_POP(y);
21       ...y...z...;
22     }
23     return;
24   }
```

図 4.3　GC ルート退避処理が挿入された関数 f の C コード

最後に，ノード C について考える．GC_PUSH/GC_POP を挿入すべき位置は C1 である．この位置で生存変数は y と z が存在するが，z は処理の対象ではないことに注意する．y が G に積まれていないため，この直前に「GC_PUSH(y);」を挿入する．また，この位置以降に triggerGC アノテーションを持つ関数の呼び出し式を含む文は存在しないため，直後に「GC_POP(y);」を挿入し，すべての変数を pop する．

以上で，すべてのノードに対する GC ルート退避処理が完了し，最終的に出力されるコードは図 4.3 のようになる．

4.5　最適化処理

前述の通り，変換した C 言語コードは C コンパイラによってコンパイルされるため，VMDLC は一般的なコンパイラが行う各種最適化を行わないが，C 言語レベルでは最適化のできない，VM 内部データ型に関する最適化処理をいくつか実装している．

4.5.1 関数のインライン展開

match 文で分岐した case 節の中で VMDL 関数が呼び出された場合，呼び出し先において再び型ディスパッチ処理が発生する場合がある．

図 3.4 の add 関数を例に挙げて説明する．この関数では，match 文による型ディスパッチの結果 25 行目の case 節が選択された場合，27 行目で to_string 関数の呼び出しが発生する．図 3.5 の to_string 関数は，add 関数の v2 を引数にとり，型ディスパッチ処理をする．結果として，to_string 関数の呼び出し側と呼び出される側の両方で同じ値に対する重複した型ディスパッチが発生している．関数のインライン展開は，このような冗長で無駄な型ディスパッチを排除する．

関数をインライン化するには，makeInline のアノテーションを付ける．このアノテーションが付与された関数は，VMDLC によってインライン展開のための情報が生成される．インライン展開は，関数の先頭から型ディスパッチ処理のみを経由して retrun 文に到達し，値を返却するものが対象である．to_string 関数を例に説明する．to_string 関数は関数の先頭で型ディスパッチ処理を行い，引数 v が String 型の場合は式 v ，Fixnum 型の場合は式 fixnum_to_string(v)，Flonum 型の場合は式 flonum_to_string(v)，Special 型の場合は式 special_to_string(v) の値を返却する．この情報を記録しておき，この関数を呼び出す側のコンパイル時に使用する．4.2 で述べた型解析処理の例において，add 関数の 27 行目の to_string 関数の呼び出しは，型環境より引数 v2 が Fixnum 型であることが既にわかっている．したがって，この呼び出しは fixnum_to_string 関数の呼び出しに置き換えられる．以上のように，VMDLC における関数のインライン展開は型ディスパッチの最適化に特化している．一般的なインライン展開とは異なり，式 to_string(v) を式 fixnum_to_string(v2) に置き換える，といった単一の式で置換できる場合にのみインライン化をするように実装されている．これは，このインライン化の目的が冗長な型ディスパッチの削減と同時に，VM サイズの増大を避けることにあるためである．

関数のインライン展開には，冗長な型ディスパッチの削減以外の利点がある．VM が持つ型変換関数には，to_string 関数のように VM 内部データ型すべてを引数として受け付けるものと，fixnum_to_string 関数のように特定の VM 内部データ型を引数として期待するものの 2 種類がある．前者を to_y 関数，後者を x_to_y 関数と呼ぶこととする．関数のインライン展開により，プログラマは変換元のデータ型が一意に定まる文脈においても，重複する型ディスパッチのための余分なオーバーヘッドを考慮することなく to_y 関数を使用できる．これにより，VM 開発者は型環境を完全に把握する必要がなくなり，適切な x_to_y 関数を選択するしなくてもよくなるため記述性が向上し，開発者の負担を軽減できる．

4.5.2 型条件の分割

関数のインライン展開は，関数呼び出しの引数のデータ型が一意に定まらない場合には適用できない．例えば，図 3.4 の add 関数に以下のような命令スペックを与えた場合を考える．

```
add (-,Fixnum,Fixnum) accept
add (-,String,Fixnum) accept
add (-,String,Flonum) accept
add (-,_,_) unspecified
```

4.2 で用いた命令スペックと異なるのは，3 行目で String 型と Flonum 型の組み合わせも許容するようにした点のみである．この場合，add 関数の 位置 (4) の型環境は，$\{\{\mathrm{v1} \mapsto \mathtt{String}, \mathrm{v2} \mapsto \mathtt{Fixnum}\}, \{\mathrm{v1} \mapsto \mathtt{String}, \mathrm{v2} \mapsto \mathtt{Flonum}\}\}$ となり，v2 は Fixnum 型と Flonum 型 2 つのデータ型を取りうることになる．ここで，27 行目の to_string(v2) のインライン展開は，v2 が Fixnum 型の場合 fixnum_to_string(v2)，Flonum 型の場合 flonum_to_string(v2) であり，一意に定まらない．これは，25 行目の case 節が v2 に関して ffs という union 型を用いて型条件を指定しているためである．

このような場合，VMDLC は case 節が持つ型条件を分割して，union 型を個々のデータ型に分けることにより，インライン展開を可能にする．上記の例で case 節を分割すると，add 関数は図 4.4 のように，3 つの case 節に分割される．各 case 節における v2 のデータ型が一意になったため，インライン展開が可能となる．7 行目の to_string(v2) は fixnum_to_string(v2) に，7 行目の to_string(v2) は fixnum_to_string(v2) に展開され，15 行目の case 節は削除される．

ただし，case 節を分割すると case 節が増え，VMDL コード上で case 節の本体が重複してしまい，VM コードのサイズが大きくなってしまう．そのため，分割する型条件は，効果が期待できるものに限定する必要がある．そのため，VMDLC ではアプリケーション開発者からのオプションを受け付け，分割する型条件を絞り込み，関数のインライン展開が可能となる場合にのみ case 節の分割を行うように実装している．

4.6 C コードの生成

VMDL では，型解析でエラーが検出されない限り，VMDL コードから C コードを生成する．C 言語の文は，match 文と rematch 文を除いて VMDL の文から一意かつ自然に決定される．match 文と rematch 文については，VMDLC は 論文 [20] で紹介したアルゴリズムを用いて，

```
 1    (vmInstruction, triggerGC) add : (JSValue, JSValue) -> JSValue
 2    add (v1, v2) {
 3      top: match (v1, v2) {
 4        ...
 5        case (String v1 && Fixnum v2) {
 6          // {{v1 ↦ String, v2 ↦ Fixnum}}
 7          String s = to_string(v2);
 8          rematch top(v1, s);
 9        }
10        case (String v1 && Flonum v2) {
11          // {{v1 ↦ String, v2 ↦ Flonum}}
12          String s = to_string(v2);
13          rematch top(v1, s);
14        }
15        case (String v1 && Special v2) {
16          // {}
17          String s = to_string(v2);
18          rematch top(v1, s);
19        }
20        ...
21      }
22    }
```

図 4.4　case 節を分割した add 関数

switch 文と goto 文を組み合わせたコードを出力する．

4.6.1　VM 命令

図 4.5 に，4.2 節の命令スペックに基づき，図 3.4 の add 関数から生成された C 言語コードを整形したもの示す．与えられた命令スペックでは「add(-,String,String)」を accept としていないが，型解析の結果からオペランドが両方とも String 型である場合の処理が必要であると判断された．前述の通り，このコード片はスレッデッドコードに埋め込まれるための形となっていて，このコード片の手前で v1 と v2 がそれぞれ第 1，第 2 オペランドとなるよう適切に定義されている．コード片の 1 行目は add 命令のラベルを生成している．3 行目からは VMDL 関数における match 文に相当しており，ネストした switch 文にコンパイルされている．eJSVM では，VM 内部データ型を判別するために，第 1 章で説明したタグ値を使用しており，get_ptag マクロを用いて取得する．これを 4 行目と 6 行目で使用している．生成された switch 文は，まず v1 のデータ型に応じてディスパッチし，String 型の場合は 5 行目の case に進み，内側の switch 文で v2 のデータ型に応じてディスパッチする．v1 のデータ型が Fixnum 型の場合には，v2 のデータ型に関するディスパッチは不要である．これは，第 1 オペランドが Fixnum 型の場合，第 2 オペランドとの組み合わせは add (-,Fixnum,Fixnum) のみ指定されているため，第 1 オペランドが Fixnum 型であれば第 2 オペランドも Fixnum 型であると推測できるためである．9 行目，26 行目の regbase[r0] への代入は，add 命令の出力先に値を設定する処理を表している．16 行目から 20 行目までのコードは rematch 文に対応していて，変数 v1，v2 に適

```
 1  DEFLABEL(HEAD);
 2  {
 3    MATCH_HEAD_add_topAT120:
 4    switch ((unsigned int)get_ptag(v1).v) {
 5    case TV_STRING:
 6      switch ((unsigned int)get_ptag(v2).v) {
 7      case TV_STRING:
 8        {
 9          regbase[r0] = concat(v1, v2);
10        }
11        goto Ladd_EPILOGUE;
12      case TV_FIXNUM:
13      default:
14        {
15          Value s_2 = to_string(v2);
16          Value tmp0 = v1;
17          Value tmp1 = s_2;
18          v1 = tmp0; v2 = tmp1;
19        }
20        goto MATCH_HEAD_add_topAT120;
21      }
22    case TV_FIXNUM:
23    default:
24      {
25        cint s_1 = to_int(v1) + to_int(v2);
26        regbase[r0] = int_to_number(s_1);
27      }
28    goto Ladd_EPILOGUE;
29    }
30  }
31  Ladd_EPILOGUE:
```

図 4.5　add 関数から生成された C 言語コード

切な値を設定したのち，goto 文を用いて外側の switch 文の先頭へ移動している．ここで冗長な代入が発生しているものの，このような処理は C コンパイラによって除去されることを期待している．

4.6.2　組み込み関数

図 4.6 に，「array_every (_,_,_) accept」という組み込み関数に基づいて図 3.6 の array_every 関数から生成された C 言語コードを整形したものを示す．4.4 節で述べたとおり，この関数は needContext アノテーションを与えられているため，引数 context が自動的に提供される．また，builtinFunction アノテーションから，いくつかの引数が渡される．2 行目から 5 行目のコードは，builtinFuction アノテーションによって追加されたものである．eJSVM の実装では，組み込み VMDL 関数に与えられる引数（array_every におけ

る a，fn，this）は，args という VM レジスタの配列を介して与えられる．2 行目で実引数にアクセスするための args を用意し，3-5 行目で局所変数に代入している．関数本体では，int_to_number 関数や get_array_element に triggerGC アノテーションが付与されているため，GC_PUSH/GC_POP 操作が自動的に挿入されている．17 行目に VMDL 関数での match 文に対応する switch 文があり，get_htag 関数によって，オブジェクト本体のヘッダタグ値と呼ばれる値を取得し，型ディスパッチを行う．eJSVM では，組み込み関数の戻り値は，set_a マクロを用いて特別なレジスタに設定する必要があるため，VMDL 関数での return 文は 36 行目と 41 行目のようにコンパイルされる．

```
1   void array_every (Context* context, cint fp, cint na) {
2     JSValue *args = (JSValue *)(&(get_stack(context, fp)));
3     Value a = args[0];
4     Value fn = args[1];
5     Value this = args[2];
6     cint len = number_to_int(get_jsarray_length(a));
7     Value t = (na >= 2) ? this : JS_UNDEFINED;
8     cint k = 0;
9     while (k < len) {
10      if (has_array_element(a, k)) {
11        GC_PUSH(t, fn, a);
12        Value jsk = int_to_number(k);
13        GC_PUSH(jsk);
14        Value val = get_array_element(a, k);
15        GC_POP4(jsk, a, fn, t);
16        Value result;
17        switch ((unsigned int) get_htag(fn).v) {
18        case HTAGV_BUILTIN:
19        default:
20          {
21            GC_PUSH5(a, t, fn, val, jsk);
22            result = send_builtin3(t, fn, val, jsk, a);
23            GC_POP5(jsk, val, fn, t, a);
24          }
25          break;
26        case HTAGV_FUNCTION:
27          {
28            GC_PUSH5(a, t, fn, val, jsk);
29            result = send_function3(t, fn, val, jsk, a);
30            GC_POP5(jsk, val, fn, t, a);
31          }
32          break;
33        }
34        if (((result == JS_FALSE) || ((result != JS_TRUE)
35            && (to_boolean(result)==JS_FALSE)))) {
36          set_a(context, JS_FALSE);
37          return;
38        }
39      }
40      k = k + 1;
41    }
42    set_a(context, JS_TRUE);
43    return;
44  }
```

図 4.6　array_every 関数から生成された C 言語コード

5　不要機能の削減機構とプロファイラ

5.1　コードセレクタ

対象アプリケーションによっては，使用されることのない VM 命令や組み込み関数が発生することがある．それらに対しては，スペックで “unspecified” という指定だけを与える．例えば，対象アプリケーションで leftshift 命令を使用しない場合，この命令に対して以下の 1 行のみ命令スペックを与える．

```
leftshift (-,_,_) unspecified
```

コードセレクタは，このような “unspecified” と指定されたコードを除外し，必要なコードだけを VMDLC で処理するようにする．特に，組み込み関数が除外された場合，その組み込み関数を表すオブジェクトが VM ヒープ上に確保されなくなるため，VM のコードサイズが大きく削減される．さらに，プログラムの実行に使用できる VM ヒープサイズが増加する．

5.2　プロファイラ

VM 命令と組み込み関数のスペック，および型条件分割のオプションの生成を支援するために，eJSTK はプロファイラを提供している．プロファイリングは，デスクトップコンピュータを用いてフルセットの eJSVM で対象アプリケーションを実行することを想定している．

5.2.1　スペック生成用プロファイラ

eJS プロジェクトの先行研究でユニットテストを用いた型情報収集機構 [19] が提案されている．本研究では，VM 命令と組み込み関数のスペック生成を行えるように，この機構を拡張した．

このプロファイラは，実行された VM 命令や組み込み関数のオペランドや引数の名前と内部データ型のログを記録するフルセットの eJSVM である．ただし，4.3 節で述べたとおり，型変換関数は VMDLC が型解析処理に基づいてスペックを生成するため，ログに記録しない．

5.2.2　型ディスパッチプロファイラ

VM 命令について，内部の型ディスパッチ処理を実行したオペランドの名前とデータ型のログを記録することができるプロファイラを実装した．このプロファイラはスペック生成用プロファイラとは別のものであり，データ型の組み合わせ単位で型ディスパッチ処理が行われた回数のログを記録する．このログを用いて，比較的多数回分岐しているデータ型の組み合わせに関して型

条件分割最適化を適用することにより，VM サイズの肥大化を抑えながら，効率的な最適化を可能とする．

例として，評価で使用したベンチマークのひとつである Mandelbrot の add 命令に関するログを以下に示す．

```
#INSN add 2
#OPRN fixnum,fixnum 8970037
#OPRN flonum,fixnum 61888
#OPRN string,fixnum 5
#OPRN fixnum,flonum 502755
#OPRN flonum,flonum 23996010
#OPRN string,string 14
```

ログ情報は，add 命令がふたつ入力オペランドを持ち，入力の第 1/第 2 オペランドについて Fixnum 型/Fixnum 型の分岐が 8,970,037 回，Flonum 型/Fixnum 型が 61,888 回，String 型/Fixnum 型が 5 回，... というようにして，実行されたという回数情報を記録している．型条件分割最適化の適用対象となる組み合わせをどのような基準で選択するかは組み込み機器の性能などを考慮して決定すべきであるが，最も呼び出し回数の多い組み合わせのみを型条件分割の適用対象とするならば，Flonum 型/Flonum 型を選べば良い．4.5.2 節で述べたとおり，VMDLC は有効な型条件のみを分割するよう設計されているため，アプリケーション開発者は単純に型条件分割を適用したい組み合わせを選ぶだけで良い．

例えば, 図 3.4 の add 関数に関して, 今回の場合に 2 番目に呼び出しの多い Fixnum 型/Fixnum 型の組み合わせを型条件分割の条件として与えたとする．Fixnum 型/Fixnum 型の組み合わせは既に単体の case 節となっているため，VMDLC はこの組み合わせに関して何もしない．

6 評価

6.1 実験環境

本研究で提案したフレームワークを用いて，2 種類の小型デバイス環境上で動作する eJSVM を生成した．

- Raspberry Pi 3 Model B+ (RP)
- FRDM-K64F (FD)

それぞれの環境を表 6.1 に示す．両環境において，eJSVM は 32 ビット向け実装を用いた．5.2 節で述べたプロファイラを用いて，各ベンチマーク向けの最小のスペックを生成した．そのため，スペックを基に生成された eJSVM は各ベンチマーク専用のものとなる．

評価では，AreWeFastYet ベンチマーク [16] から 8 プログラム，SunSpider ベンチマーク[*1]から 12 プログラムを使用した．これらのベンチマークは eJSVM 上で動作できるように若干の変更が加えられている．また，IoT デバイスで実際に使用されているプログラムをベースに作成した dht11 プログラム [17] を使用した．このプログラムは，温湿度センサから送信されるビット列を温湿度の数値に変換する処理を繰り返すものである．

eJSVM は，各ベンチマーク毎に以下の 4 種類の VM を使用した．

Prev: VM 命令を記述するための先行研究の DSL (vmgen) を用いたもの．

Opt⁻: VMDL を使用し，最適化なしのもの．

表 6.1 実行環境

	RP	FD
CPU	Cortex-A53 (ARMv8) 64-bit SoC @ 1.40 GHz	Cortex-M4F (ARM) @ 120 MHz
メモリ	1 GB	256KB RAM + 1,024 KB flash
OS	Raspbian 9.13	
C コンパイラ	GCC 6.3.0 20170516 (Raspbian 6.3.0-18+rpi1+deb9u1)	GCC 7.3.1 20180622 (15:7-2018-q2-6)

[*1] https://webkit.org/perf/sunspider/sunspider.html

$\mathsf{Opt}^{\mathsf{I}}$: VMDL を使用し，インライン展開最適化を施したもの．

$\mathsf{Opt}^{\mathsf{IS}}$: VMDL を使用し，インライン展開および型条件件分割最適化を施したもの．

特に型条件件分割最適化は，5.2.2 節で述べたプロファイラによって得られたデータから，各命令に関して平均回数以上実行された組み合わせを分割の対象とした．実行時の VM ヒープサイズは 128 キビバイトとした．また，eJSVM が持つ最適化オプションのインラインキャッシュ [5]，アロケーションサイトキャッシュ [4] は無効とした．FD 上で eJSVM を実行するためにメモリサイズに制限があったためである．また，GC のアルゴリズムは，Fusuma コンパクション [18] を使用した．

6.2 記述性と安全性

VMDL を用いることで，VM プログラムの記述性と安全性が向上した．この節では開発における実例を挙げる．

6.2.1 型ディスパッチ処理

3.5 節の VMDL による記述例や 4.6 節のコンパイル例で述べたように，C 言語を用いて複数のデータ型に基づく型ディスパッチ処理を記述する場合，ネストされた複雑な `switch` 文によって記述される．これに対して，VMDL では `match` 文によって複数のデータ型に基づく型ディスパッチ処理をネストなく記述できる．データ型の条件も論理演算を用いた簡潔な表現で記述できる．

6.2.2 GC ルート退避処理の自動挿入

VMDL 関数に与えられたアノテーションは VMDLC に情報を提供し，VMDLC は VM の内部実装に沿った C プログラムを自動で生成する．この仕組みによって，VM 開発者は VM の詳細な実装を気にせずに VM コードを記述でき，また，VMDL 関数を簡潔に記述できる．4.4.1 節で述べた `triggerGC` アノテーションによる GC ルート退避処理の自動挿入は，記述性向上の典型的な例である．

従来の eJSVM (Prev) の VM コードにおいては，88 組の `GC_PUSH`/`GC_POP` 処理が VM 命令，組み込み関数，型変換関数に手動によって含まれていた．これらの命令，関数が VMDL 関数に置き換えられた結果，88 組の `GC_PUSH`/`GC_POP` 処理すべてについて，手動による明示的な挿入が不要となった．これは，VM コードの記述性を向上させ，VM 開発者の負担を軽減する効果があることを示している．また，必要なコードの挿入を忘れるというミスも無くなり，VM コードの安全性向上にも貢献している．

6.2.3 コンテキスト情報の引き渡し

3.2.1 節で述べたように，needContext アノテーションはコンテキスト情報を渡す必要のある関数に付与する．このアノテーションが付与された関数の C コードを生成する際に，VMDLC は自動的に context の引数を与える．これにより，VM 命令，型変換関数，組み込み関数を C 言語で直接記述する際に必要であった context 引数の明示が必要なくなる．実際，Prev で明示的に context を渡す 466 箇所の記述は，VMDL を用いることですべて削減された．

6.2.4 型検査

VM 内部データ型に対する型検査は，型エラーを静的に検出するのに役立つ．C 言語の場合，1.2 節で述べたように，Value 型は対象言語のすべての第一級のデータ型の値を表す唯一の C 言語データ型であり，VM 内部データ型に基づく静的な型検査は不可能である．これに対して，VMDL コードは VM 内部データ型の粒度で静的な型検査を行うことができる．これにより，VM プログラムの安全性が向上する．

具体的な例として，String.prototype.split メソッド[*2]の実装における事例を挙げる．組み込み関数 split(separator, limit) は，レシーバオブジェクトである文字列を separator で与えられた文字列（あるいは正規表現オブジェクト）で分割した部分文字列を格納した配列オブジェクトを返却する．第 2 引数の limit は，返却する配列の長さの上限を指定する．Prev では，この組み込み関数を C 言語で定義しており，その中には次のようなコードがあった．

```
cint lim = (args[2] == JS_UNDEFINED) ? ... : number_to_cint(args[2]);
```

ここで，args[2] は limit で，そのデータ型は Value であり，cint は C 言語データ型の符号付き整数の型シノニムである．しかし，上記のコードは，split メソッドの使用上，limit は任意の VM 内部データ型が許容されているにもかかわらず，number_to_cint 関数の引数は VM 内部データ型の Number 型（Fixnum 型あるいは Flonum 型）に制限されているという，バグとなっていた．このバグは，引数の型が Value 型であったため，C コンパイラは検出することが不可能であった．実際に，後述の VMDL による実装を VMDLC がコンパイルする際にエラーを検出するまで，このバグは見落とされていた．

次に，VMDL 関数として，split メソッドを定義した．当初，関数は図 6.1 の (a) に示すように記述した．これは C 言語による実装に直接対応するものであった．しかし前述の理由により，VMDLC はこのコードをコンパイルする際に，型エラーを報告した．これは，2 行目で number_to_cint 関数の引数が Number 型として宣言されているのに対して，渡された引数 limit は 4 行目で JSValue 型として宣言されており，データ型の不整合が起こったためである．

[*2] https://262.ecma-international.org/5.1/#sec-15.5.4.14

```
1  union Number = Fixnum || Flonum
2  externC function number_to_cint : Number -> int
3  ...
4  (triggetGC, builtinFunction) string_split : (String, JSValue, JSValue) -> JSValue
5  string_split (s, separator, limit) {
6    ...
7    int lim = (limit == JS_UNDEFINED) ? ... : number_to_cint(limit);
8    ...
9  }
```

(a) 修正前の定義

```
1  union Number = Fixnum || Flonum
2  externC (triggerGC) function toInteger : JSValue -> int
3  ...
4  (triggetGC, builtinFunction) string_split : (String, JSValue, JSValue) -> JSValue
5  string_split (s, separator, limit) {
6    ...
7    int lim = (limit == JS_UNDEFINED) ? ... : toInteger(limit);
8    ...
9  }
```

(b) 修正後の定義

図 6.1　VMDL による split 関数の定義

ここでの不整合は，JSValue 型が Number 型のサブセットとなっていないことである．そこで，関数を図 6.1 の (b) に示すように修正した．このコードでは，number_to_cint 関数の代わりに，JSValue 型を引数とする toInteger 関数を使用している．この修正により，split メソッドは正しく定義された．

6.3　効率性

表 6.2 に RP と FD 向けに生成された各 VM のサイズを，表 6.3 にインライン展開最適化の適用回数を示す．ここでは，比較のために Prev を拡張して，組み込み関数のコード選択機構を含めた Prev^{+} を実装した．表 6.2 の表の各 $\mathsf{Opt}^{\mathsf{IS}}$ の列は対応する $\mathsf{Opt}^{\mathsf{I}}$ の列とのサイズ差を表し，空の列は差がないことを表している．VM サイズに差が生じた理由は，型条件分割最適化によって case 節の数が増加したためである．case 節が分割されると，インライン展開最適化の適用回数が増える．表 6.3 の $\mathsf{Opt}^{\mathsf{IS}}$ の列では括弧内に型条件分割最適化の適用回数を示している．

Prev および Prev^{+} では，VM 開発者がコード記述時に手動で最適化を行っている．これに対し，VMDL を使用することで，VM 開発者はそのような手動による最適化を行う必要がなくなる．その上で，RP の $\mathsf{Opt}^{\mathsf{I}}$ と $\mathsf{Opt}^{\mathsf{IS}}$ の VM サイズは Prev^{+} より若干小さく，FD の VM サイズは Prev^{+} より若干大きくなっており，大きなサイズ的オーバーヘッドは見られなかった．

Opt^{-} と $\mathsf{Opt}^{\mathsf{I}}$ を比較すると，インライン展開最適化は VM サイズに影響を与えていないこと

表 6.2　VM サイズ

Program	RP					FD				
	Prev	Prev$^+$	Opt$^-$	Opt^{I}	Opt^{IS}	Prev	Prev$^+$	Opt$^-$	Opt^{I}	Opt^{IS}
AreWeFastYet benchmark										
Bounce	77,876	61,396	60,060	59,964		138,744	101,456	103,440	103,184	
List	77,732	61,232	59,804	59,780		138,680	101,440	103,280	103,088	
Mandelbrot	78,212	61,772	60,520	60,344	+166	139,216	101,440	104,048	103,912	+312
Permute	77,820	61,324	59,884	59,856		138,752	101,440	103,360	103,160	
Queens	77,760	61,264	59,828	59,820		138,688	101,440	103,280	103,088	
Richards	78,324	61,828	60,556	60,548		139,432	101,440	104,056	103,824	
Sieve	77,792	61,296	59,868	59,860		138,712	101,440	103,328	103,136	
Towers	77,912	61,424	59,988	59,972		138,816	101,440	103,424	103,224	
SunSpider benchmark										
3d-cube	78,044	61,220	60,472	60,188	+ 20	139,232	107,224	110,048	109,880	+224
access-binary-trees	76,860	59,800	58,208	58,088		138,304	101,104	101,776	101,744	
access-fannkuch	76,212	60,100	57,628	57,556		137,480	102,512	101,536	101,504	
access-nbody	77,708	60,276	59,172	59,044	+132	139,000	101,624	103,728	103,608	+328
bitops-3bit-bits-in-byte	75,808	58,360	56,548	56,516		136,968	100,720	100,152	100,112	
bitops-bits-in-byte	75,736	58,288	56,456	56,436		136,928	100,720	100,112	100,072	
bitops-bitwise-and	75,704	58,256	56,472	56,468		136,928	100,720	100,096	100,104	
controlflow-recursive	75,852	58,404	56,668	56,600		137,096	100,720	100,288	100,248	
math-partial-sums	76,748	59,924	58,448	58,312	+ 68	138,120	111,072	111,880	111,768	+240
math-spectral-norm	77,076	60,228	59,044	58,908	+ 28	138,384	101,624	103,088	102,960	+ 72
string-base64	77,116	61,252	60,088	59,644	+100	138,320	101,864	103,320	102,832	+ 64
string-fasta	77,400	60,912	59,664	59,504	+140	139,176	101,312	104,248	104,152	+ 72
IoT program										
dht11	76,816	60,200	58,864	58,772		137,832	101,040	101,888	101,504	

（単位：bytes）

がわかる．これは，関数呼び出し式から別の式へのインライン化に制限したためである．これにより，展開後のコードサイズの増加を抑え，更に，展開後にどこからも呼び出されなくなる型変換関数を排除することができた．

型条件分割最適化を行ったすべてのプログラムにおいて，新しい case 節が発生するため，Opt^{IS} の VM サイズは Opt^{I} よりも増加した．しかし，その増加量は RP で 166 バイト，FD で 328 バイトとわずかであり，VM サイズ全体の 0.3% 程度であった．このようなごく小さなサイズ的オーバーヘッドであるにも関わらず，次節で述べるように，このようなプログラムに対する Opt^{IS} の実行時間は平均して Opt^{I} よりも 5% ほど短い結果となった．

表 6.3　インライン展開の適用回数

ベンチマークプログラム	インライン展開回数	
	$\mathsf{Opt}^{\mathsf{I}}$	$\mathsf{Opt}^{\mathsf{IS}}$
AreWeFastYet benchmark		
Bounce	22	
List	20	
Mandelbrot	24	+5 (3)
Permute	20	
Queens	20	
Richards	25	
Sieve	20	
Towers	20	
SunSpider benchmark		
3d-cube	33	+4 (2)
access-binary-trees	17	
access-fannkuch	18	
access-nbody	18	+6 (3)
bitops-3bit-bits-in-byte	14	
bitops-bits-in-byte	12	
bitops-bitwise-and	12	
controlflow-recursive	14	
math-partial-sums	23	+4 (2)
math-spectral-norm	22	+2 (1)
string-base64	33	+1 (1)
string-fasta	28	+3 (2)
IoT program		
dht11	27	

6.3.1　実行速度

VMDL を使用することで，VMDLC が型解析と最適化を行うため，VM 開発者は手動による最適化を行う必要がなくなる．Prev では開発者による慎重なコーディングが求められていたものの，VMDL はそれ無しに Prev と同等，あるいはそれ以上に効率的な VM を生成することができた．

図 6.2 と図 6.3 にそれぞれ RP と FD におけるベンチマークプログラムの実行時間を示す．ここでは，Opt^{-} と $\mathsf{Opt}^{\mathsf{I}}$ の結果のみを示し，$\mathsf{Opt}^{\mathsf{IS}}$ の結果については後述する．実行時間はそれぞれのベンチマークと実装の組み合わせ毎に VM を 5 回ずつ生成し，RP は 10 回，FD は 2 回実行した平均値である．各プログラムの実行時間は Prev の実行時間を 1 として正規化して示している．また，幾何平均も示している．なお，Richards と controlflow-recursive はスタック領域

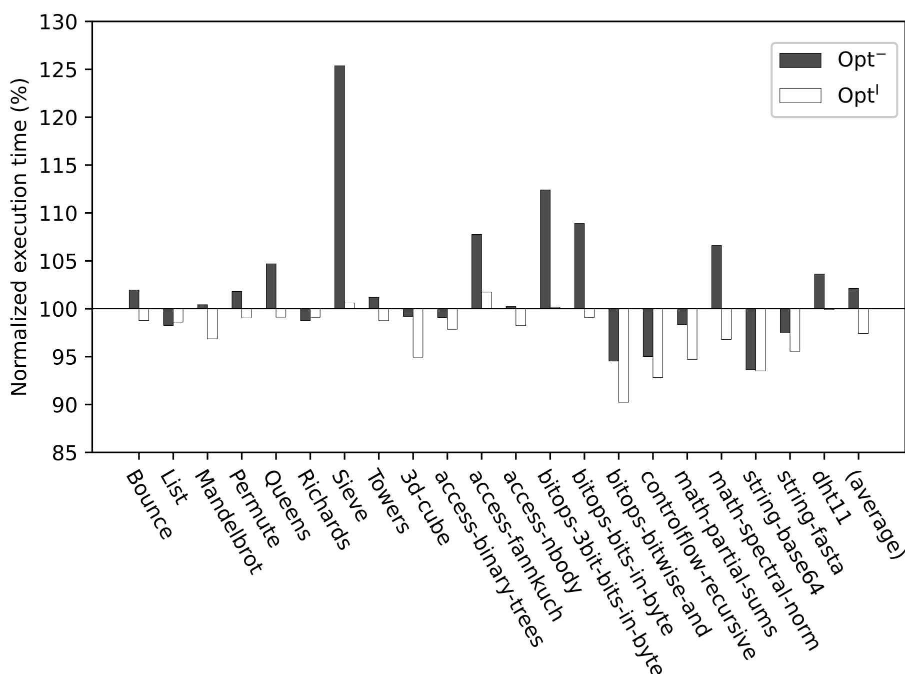


図 6.2　正規化された実行時間 (RP)

の不足などの理由により，FD 上では実行できなかったため FD に関しては実行時間を示していない．

いくつかのプログラムにおいて，Opt^{-} は Prev よりも遅かった．これは，Prev のコードでは変数が取りうる VM 内部データ型の文脈に合わせて VM 開発者が手動で最適化していたためである．4.5.1 で述べたように，C 言語（および vmgen）を用いた実装では各所で適切な `x_to_y` 関数を使用していた．一方で，VMDL では型変換関数に関してより抽象的な `to_y` 関数を使用していたため，Opt^{-} では最適化されず，冗長な型ディスパッチ処理が発生してしまった．Opt^{-} の全ベンチマークの結果を平均すると，実行時間は Prev の 102.1% (RP)，101.1% (FD) であった．

$\mathsf{Opt}^{\mathsf{I}}$ では，Prev とほぼ同等の実行時間か，最大で 10% ほど短くなった．これは，VMDL コードが関数のインライン展開によって VM 内部データ型に関して最適化されたことを意味している．また，$\mathsf{Opt}^{\mathsf{I}}$ の全ベンチマークの結果を平均すると，実行時間は Prev の 97.4% (RP/FD) であった．

型条件分割最適化が適用された 7 つのベンチマークでは，$\mathsf{Opt}^{\mathsf{IS}}$ の実行時間が $\mathsf{Opt}^{\mathsf{I}}$ の実行時間よりも短かった．図 6.4 に $\mathsf{Opt}^{\mathsf{I}}$ 実行時間を 1 とした実行時間の比率を示す．RP と FD の両方で同じ傾向を示し，平均すると 95.6% (RP)，96.2% (FD) であった．

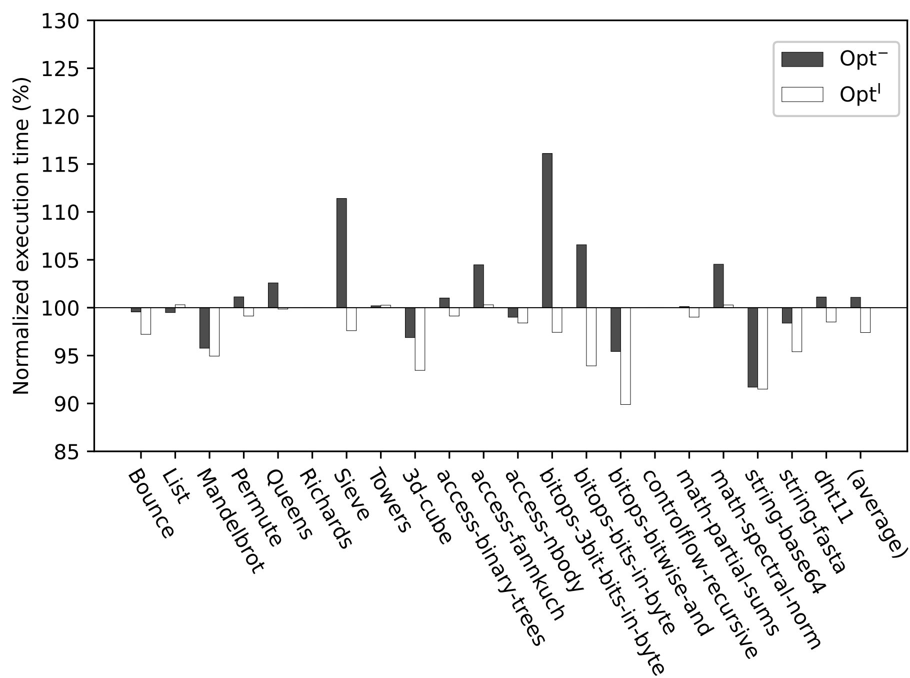
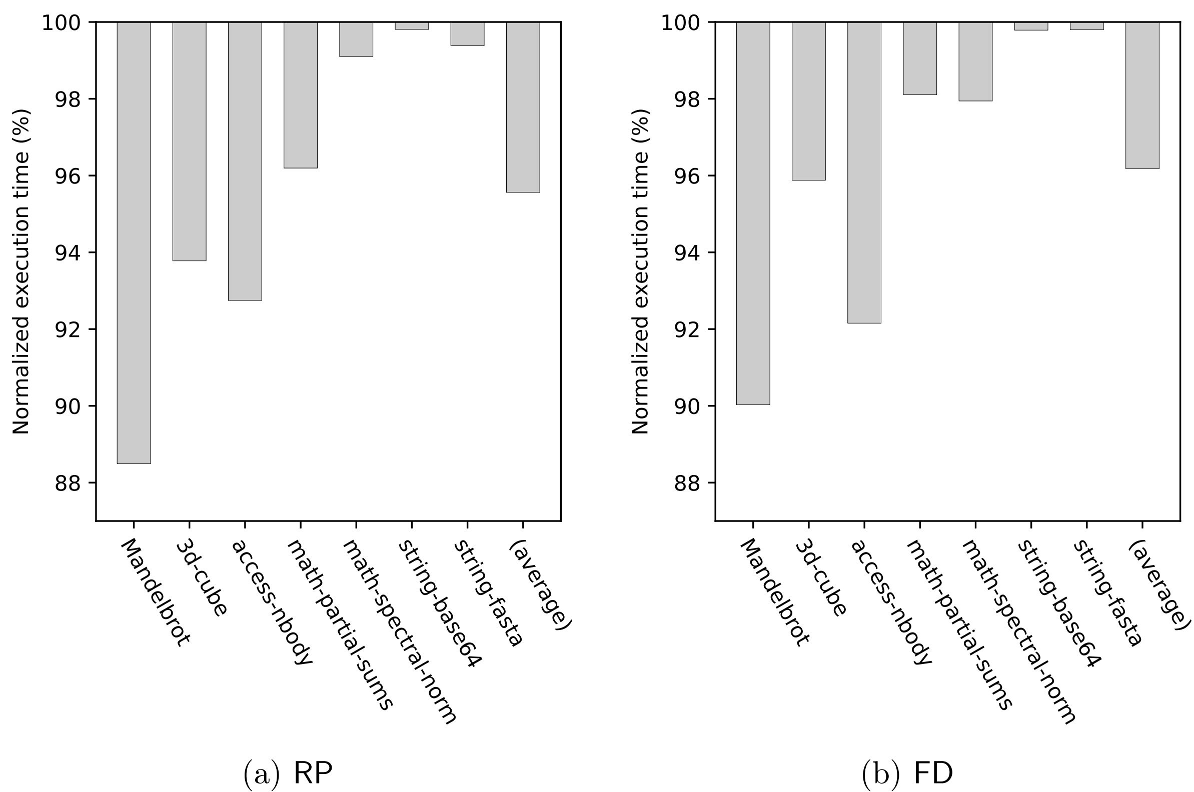


図 6.3　正規化された実行時間 (FD)



(a) RP　　(b) FD

図 6.4　$\mathsf{Opt}^{\mathsf{I}}$ に対する $\mathsf{Opt}^{\mathsf{IS}}$ の実行時間

7 関連研究

eJS では以前，VM 命令を記述するために単純な DSL である vmgen [20] を使用していた．構文は限定的で，VM 命令の定義のオペランドのデータ型に基づいて型ディスパッチ処理を記述するために使われていた．命令はオペランドのデータ型に対応する操作の集合として記述され，各操作は C 言語のコード片であり，生成されるコードに直接埋め込まれる．DSL 処理系の役割は，`switch` 文を用いた型ディスパッチコードを生成することである．VMDL は，VM 命令，型変換関数，組み込み関数を記述できる構文を提供しており，また，コードに対して静的な型解析が可能となっている．

Latendresse [14] は，新しい VM 命令を自動生成することにより，コンパクトな Scheme 言語用の VM を生成した．Latendresse のアプローチでは，繰り返し出現する命令列に対して特別な新しい命令を割り当てることで VM サイズを小さくする．eJS のアプローチは全く異なっており，対象アプリケーションに不要な機能を除去することで VM サイズを小さくする．

Jikes RVM [1] は高級言語である Java を用いて Java VM を実装している．VM は自己ホスト化されており，C や C++ で記述されている Java プラットフォーム用の VM に比べて，移植性などの面で利点を持つ．Maxine VM [21] も同じく Java を用いて Java VM を実装している．Jikes RVM の AoT コンパイラは，Java VM のクラスファイルをネイティブコードにコンパイルする．このとき，GC のためのライトバリアが AoT コンパイラによって VM の実行ファイルに配置される．AoT コンパイラは，GC に関するアノテーションがメソッドに付与されているかどうかでコンパイルを制御する．GC に関する処理についてアノテーションを用いる点では VMDL と同じである．

Vmgen [*1] [8][9] とその後継の Tiger [3] は，スタックマシンを特別にサポートした，仮想マシンインタプリタ生成ツールである．Gforth [7] や Cacao Java VM [10] といった VM インタプリタは Vmgen を用いている．これらの VM では，DSL を定義しており，基本的には C 言語のコード片にメタ情報を付与したものとなっている．メタ情報は，合成命令などの一般的な最適化を行ったインタプリタを生成するために利用されている．VMDL とは異なり，Vmgen と Tiger では動的に型付けされる値をサポートしていない．

Truffle DSL [11] は，AST インタプリタを記述するための DSL で，Java 言語に特殊なアノテーションを付与したものである．Truffle では，`add` などの対象言語における操作は AST ノードとして記述される．VM 開発者は与えられた引数に対しての操作をメソッドの形で実装する．Truffle DSL では，VM 開発者が特定の引数の型の組み合わせに特化した，特殊化されたメソッドを定義することができる．Truffle のフレームワークは引数のデータ型に基づき，特殊化され

[*1] eJS で利用していたものとは異なる．

たメソッドを実行するようにディスパッチするコードを生成する．Truffle DSL で記述されたAST インタプリタは，特殊な JIT コンパイラ [23] を持つ Java VM によって実行される．

Hwu ら [12] は，C 言語プログラムに関するインライン展開最適化の手法を提案した．C 言語プログラムに対する一般的なインライン展開による最適化について，形式的に手法を述べている．インライン展開を行う際に考慮すべき問題について議論しており．プログラムのコードサイズや手続きの呼び出し回数についての近似的な評価方法について提案している．VMDL では，現状インライン展開の対象が型変換関数のみであり，また，一般的なインライン展開による最適化は C コンパイラにより行われることを期待しているため，VMDLC はこの論文で提案しているようなインライン展開を行っていない．

Lee ら [15] は，関数の分割による部分的なインライン展開最適化の手法を提案した．対象の関数の制御フローグラフに対して部分をグラフを構築し，インライン化およびアウトライン化することによって，インライン展開のメリットである実行速度の向上と，デメリットであるコードの肥大化の回避の両立を図っている．速度向上の効果やアウトライン化された関数の呼び出しにかかるオーバーヘッドから数値的にインライン化する部分を計算する．VMDL では，型変換関数の返却式のみを展開するため，コード肥大化の評価は行っていない．

8 おわりに

本研究では，組込みシステム向けの VM プログラムを，VM のデータ型の粒度で静的に解析できる DSL である VMDL を用いて記述し，型に基づく最適化を適用した C 言語にコンパイルする手法を提案した．この提案は，組込みシステム用 JavaScript VM である eJSVM を開発するためのフレームワークとして実装されている．このフレームワークには，VMDL，VMDLC (VMDL コンパイラ)，および関連するツール群が含まれている．VMDL は，C 言語で直接記述された VM プログラムでは困難であった，VM 内部データ型に基づく VM プログラムの記述と VMDLC による VM 内部データ型粒度での解析を可能にし，1 章で述べた記述性，安全性，効率性の問題を改善した．

謝辞

本研究を行うにあたり，終始ご指導を賜りました岩崎英哉教授に深く感謝いたします．また，多くの助言をいただきました東京大学の鵜川始陽准教授にも感謝いたします．また，多くの面でお世話になった岩崎研究室の皆様，特に多くの場面で相談に乗っていただいた小野澤拓氏にも御礼申し上げます．

参考文献

[1] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.*, 44(2):399–418, 2005.

[2] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

[3] Kevin Casey, David Gregg, and M. Anton Ertl. Tiger - an interpreter generation tool. In Rastislav Bodík, editor, *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3443 of *Lecture Notes in Computer Science*, pages 246–249. Springer, 2005.

[4] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. Memento mori: dynamic allocation-site-based optimizations. In Antony L. Hosking and Michael D. Bond, editors, *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA, June 13-14, 2015*, pages 105–117. ACM, 2015.

[5] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, editors, *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 297–302. ACM Press, 1984.

[6] ECMA International. *Standard ECMA-262 - ECMAScript 2021 Language Specification.* 2021.

[7] M. Anton Ertl. A portable forth engine. In *Proc. EuroFORTH '93 Conference*, 1993.

[8] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen - a generator of efficient virtual machine interpreters. *Softw. Pract. Exp.*, 32(3):265–294, 2002.

[9] David Gregg and M. Anton Ertl. A language and tool for generating efficient virtual machine interpreters. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2003.

[10] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an efficient java in-

terpreter. In Louis O. Hertzberger, Alfons G. Hoekstra, and Roy Williams, editors, *High-Performance Computing and Networking, 9th International Conference, HPCN Europe 2001, Amsterdam, The Netherlands, June 25-27, 2001, Proceedings*, volume 2110 of *Lecture Notes in Computer Science*, pages 613–620. Springer, 2001.

[11] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. In Ulrik Pagh Schultz and Matthew Flatt, editors, *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, pages 123–132. ACM, 2014.

[12] Wen-mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. pages 246–257, 1989.

[13] Takafumi Kataoka, Tomoharu Ugawa, and Hideya Iwasaki. A framework for constructing javascript virtual machines with customized datatype representations. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1238–1247. ACM, 2018.

[14] Mario Latendresse. Automatic generation of compact programs and virtual machines for scheme. In *Proc. Workshop on Scheme and Functional Programming*, September 2000.

[15] Junpyo Lee, Jae-Jin Kim, Soo-Mook Moon, and Suhyun Kim. Aggressive function splitting for partial inlining. In *15th Workshop on Interaction between Compilers and Computer Architectures, INTERACT 2011, San Antonio, Texas, USA, February 12, 2011*, pages 80–86. IEEE Computer Society, 2011.

[16] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-Language Compiler Benchmarking—Are We Fast Yet? In *Proc. 12th Symposium on Dynamic Languages (DLS 2016)*, pages 120–131. ACM, November 2016.

[17] Hiro Onozawa, Hideya Iwasaki, and Tomoharu Ugawa. Customizing JavaScript virtual machines for specific applications and execution environments (in japanese). *Computer Software*, 38(3):23–40, 2021.

[18] Hiro Onozawa, Tomoharu Ugawa, and Hideya Iwasaki. Fusuma: double-ended threaded compaction. In Zhenlin Wang and Tobias Wrigstad, editors, *ISMM '21: 2021 ACM SIGPLAN International Symposium on Memory Management, Virtual Event, Canada, 22 June 2021*, pages 94–106. ACM, 2021.

[19] Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. Collecting type information using unit tests for customizing javascript virtual machines. In *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented*

Languages, Programs and Systems, ICOOOLPS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. ejstk: Building javascript virtual machines with customized datatypes for embedded systems. *J. Comput. Lang.*, 51:261–279, 2019.

[21] Christian Wimmer, Michael Haupt, Michael L. Van de Vanter, Mick J. Jordan, Laurent Daynès, and Doug Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, 2013.

[22] Allen Wirfs-Brock and Brendan Eich. Javascript: the first 20 years. *Proc. ACM Program. Lang.*, 4(HOPL):77:1–77:189, 2020.

[23] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 662–676. ACM, 2017.

[24] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 187–204. ACM, 2013.