Regular Paper

Generating Virtual Machine Code of JavaScript Engine for Embedded Systems

Yuta Hirasawa[1,†1,a)] Hideya Iwasaki[1,†2,b)] Tomoharu Ugawa[2,c)] Hiro Onozawa[1,†3,d)]

Received: February 1, 2022, Accepted: June 14, 2022

Abstract: Virtual machines (VMs) for dynamically managed languages such as JavaScript are generally implemented in C or C++. Implementation of VMs in such low-level languages offers the advantage of high flexibility, but it suffers from problems of descriptiveness and safety. These problems are due to the fact that even though a variety of VM operations are based on the VM's internal datatypes for first-class objects in the target language, the C code typically treats all VM internal datatypes as a single type in C. In addition, VMs implemented in C or C++ have a size problem which is a serious issue for VMs on embedded systems. The reason for this problem is the difficulty of eliminating unnecessary code fragments from the C code for a specific application. To solve these problems, we propose a domain-specific language for describing VM programs and a corresponding compiler to C programs, called VMDL and VMDLC, respectively. We also propose related utility tools. This framework enables generation of C source code for an eJSVM, a JavaScript VM for embedded systems. VMDL's concise syntax enables static VM datatype analysis for optimizations and error detection. We evaluated the framework and confirmed its effectiveness from both qualitative and quantitative viewpoints.

Keywords: JavaScript virtual machine, domain-specific languages, VM internal datatype, compilers

1. Introduction

JavaScript [23] is widely used for developing not only web applications on browsers but also server-side programs and even applications on embedded systems. Virtual machines (VMs) for dynamically managed languages such as JavaScript are typically implemented in C or C++ [25]. Implementations in such languages have the advantage of high flexibility, because these languages enable programmers to describe low-level processing. Despite this advantage, the use of C [*1] to implement VMs has the following three problems. The first two are problems for the VM developer, while the last one is a problem for the application developer.

Descriptiveness problem. Writing VM code directly in C tends to lead to complex code, which in turn decreases the descriptiveness of the VM code.

For example, a VM for a dynamic language contains many dispatch operations, each of which identifies the datatypes of given values at runtime and selects an appropriate branch. We call this operation a *type dispatch*. In C, type dispatches are typically described in terms of `switch` statements for all possible combinations of datatypes. However, this makes the code complicated, especially when type dispatches depend on multiple values and nested `switch` statements are used. Furthermore, in the case of JavaScript, a VM sometimes performs type dispatch again after performing type conversions, which makes the process even more complex.

Another example of the descriptiveness problem is the insertion of patterned code fragments. A VM developer often has to insert code fragments for managing the root set of garbage collection (GC), write barriers, and so on. Suppose that a VM uses a GC algorithm that moves objects. When calling a C function that may invoke GC, the addresses for the values of the GC target must be saved to the GC root area before the function call so that the GC can find the locations of those values. For example, if two variables `a` and `b` hold pointers to GC target objects and a C function `g` may cause GC, it is necessary to insert code fragments to push the addresses of `a` and `b` and to pop them after the call to `g`, as follows.

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

Forgetting such `push` and `pop` operations will cause `a` and `b` to become dangling pointers if GC happens to occur during the execution of `g`. Unfortunately, the compiler cannot detect this event. Furthermore, the insertion of these code fragments is a tedious and error-prone task, and the fragments also decrease maintainability. Accordingly, this is not only an instance of the descriptiveness problem but also of the safety problem that is described next because dangling pointers will cause a VM to crash.

Safety problem. C also involves the risk of overlooking datatype-related bugs that cannot be detected at compile time. In a typical implementation, the C code for a VM defines a single

[1] The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[2] The University of Tokyo, Bunkyo, Tokyo 113–0001, Japan
[†1] Presently with Fujitsu Limited
[†2] Presently with School of Science and Technology, Meiji University
[†3] Presently with KLab Inc.
[a)] 19yhirasawa@ipl.cs.uec.ac.jp
[b)] hideya.iwasaki@acm.org
[c)] tugawa@acm.org
[d)] 19honozawa@ipl.cs.uec.ac.jp

[*1] For simplicity, we write "C" to refer to C/C++.

type (say, `Value`) that corresponds to *all first-class data* in the target language. The individual datatype is typically identified by using some part within the value of a `Value` type as a tag. In fact, internal representations that use tags can be found in many programming language implementations.

In dynamic languages, datatypes are determined at runtime. If a C function expects its argument to be of a specific datatype in the target language, an `assert` macro is often used. Suppose that the C datatype, `Value`, is implemented as a type synonym for `uint64_t`, and that the lowest three bits are used as a tag. Suppose also that the datatype `Fixnum` for fixed-length integers in the target language is represented by `Value` with a specific tag value. If the C function `f` expects only a `Fixnum` as its actual argument, then `f`'s definition would appear as follows.

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

Here, `isFixnum` is a C function (or macro) that determines whether the tag of a given `Value` represents a `Fixnum`. Because the checking by `assert` is done at runtime, a call to `f` with a non-`Fixnum` actual argument may be overlooked even if the VM developer performs comprehensive testing.

Another example of a bug that cannot be detected statically in C is conversion from a `Value` to a C datatype. A computation on a `Fixnum` object `n` should be done after converting `n` to the C datatype `int64_t` as follows.

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

However, a C compiler will not report an error if the VM developer forgets the shift operation to drop the tag.

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

As a result, such bugs are likely to be overlooked.

VM size problem. This problem is particularly serious for VMs on embedded systems. Because the installed memory on an embedded system is limited, it is important to reduce the VM size to handle only the minimum necessary capability. Generally, an application on an embedded system is designed for a specific purpose, such as measuring temperature and humidity data and sending it to a server, and the application is executed repeatedly. Accordingly, a VM for an embedded system does not have to be fully featured but only needs the minimum datatypes, VM instructions, and built-in functions required to run the target application. However, a VM written in C, where `switch` statements are used for type dispatches, may include `case` branches that are never taken by the target application. The VM might also include instructions and built-in functions that are unnecessary for the target application. Furthermore, some patterned code fragments such as those for GC root management may be redundant; for example, if a variable always has a `Fixnum` value, then it does not need to be saved to the GC root area.

To solve these three problems, we propose an approach to describe VM programs for embedded systems in a domain-specific language (DSL) and to compile them into C programs. We designed the DSL to perform static analysis of possible datatypes for each variable of type `Value` on the basis of the given operand datatypes of VM instructions and the argument datatypes of functions. Through the results of this analysis, the compiler selects only the necessary parts of the DSL programs and translates them into C programs. In this translation process, the compiler inserts patterned code fragments that are judged to be necessary, and it generates an efficient type dispatch code. The datatypes of operands and arguments for a specific application can be obtained through a profiling run of the application on a desktop computer.

By following the above approach, we have implemented a framework to generate the C source code of an eJSVM [21], which is a JavaScript VM for embedded systems. Although our targets are JavaScript and C, the underlying idea is general and not limited to those languages.

Our proposed approach has the following three main components.

- *VMDL* (Virtual Machine Description Language), a statically typed DSL for the VM developer to describe eJSVM code.
- *VMDLC* (VMDL Compiler), a compiler that applies datatype analysis on VMDL code to generate the minimum required C source code fragments for an eJSVM that is specialized for the target application. This component is used by the application developer.
- *A set of utility tools* to help generate the customized eJSVM. The tools include a *code selector*, which selects only the necessary function definitions written in VMDL according to the *specifications* of the target application; and a *specification generator*, which generates specifications from the results of a profiling execution. These tools are also used by the application developer.

Note that VMDL is not designed to have the same level of descriptive power as the C language. It excludes features such as pointers and the capabilities of handling the raw bit patterns of `Value`s, because these makes it difficult to perform datatype analysis. It does not provide less frequently used features, either. When a VM requires such features, the VM developer can describe them via C functions and call them from the VMDL code. In addition, VMDL does not provide tightly optimized assembly code generation; instead, it leaves common optimizations to the C compiler.

The organization of this paper is as follows. Section 2 gives an overview of eJS which is the target of the proposed framework. Section 3 describes the design of VMDL mainly through specific examples. Section 4 describes VMDLC, including its optimizations, and Section 5 describes the Utility tools. Section 6 describes evaluations of the framework with some obtained experiences and some experimental results on two kinds of processors. Finally, we discuss related work in Section 7, and conclude the paper in Section 8.

2. System Overview

2.1 eJSVM

The proposed framework generates eJSVM code for a target application. Thus, we first explain the eJSVM, which is a compact, efficient JavaScript VM that is based on eJS (embedded JavaScript) [21]. eJS enables application development for embedded systems by using a high-level language, JavaScript. The eJSVM supports a subset of ECMAScript 5.1, which excludes complicated features such as the `eval` function. Given the background described for the "VM size problem" in Section 1, eJS is focused on the capability of generating customized eJSVMs,

Table 1 eJSVM internal datatypes.

JS datatypes	VM internal datatypes
Undefined, Null, Boolean	`Special`
String	`String`
Number	`Fixnum`, `Flonum`
Object	`Simple_object`
Array	`Array`
Function	`Function`, `Builtin`
Regexp	`Regexp`

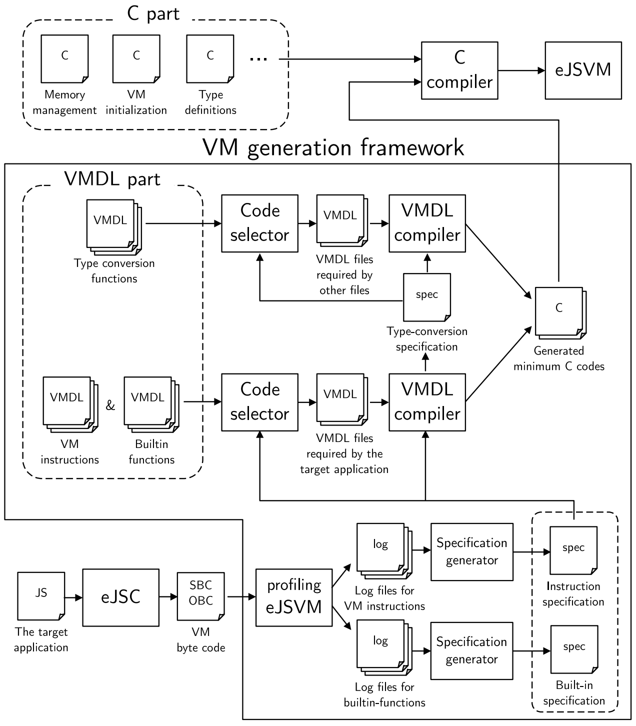


Fig. 1 Overview of the eJSTK.

which are specialized for specific target applications.

The eJSVM is a register-based VM, and its VM instruction set was originally designed in the eJS research. We refer to the datatypes of the first-class values inside an eJSVM as *VM internal datatypes*, and to the datatypes defined in the JavaScript specification [6] as *JS datatypes*. **Table 1** shows the corresponding items among the JS datatypes and VM internal datatypes.

2.2 eJS Toolkit

The *eJSTK* (eJS ToolKit) is provided as a toolkit for generating customized eJSVMs according to *specifications* for the target application. The proposed framework is a main part of the eJSTK. **Figure 1** illustrates the overall structure of the eJSTK. Although the eJSTK provides capabilities for customizing the internal representations of VM internal datatypes and generating a JavaScript compiler (eJSC) for an eJSVM, these capabilities are omitted from the figure because due to being outside the scope of this paper.

The VM source code consists of programs in C, which we call the *C part*, and those in VMDL, which we call the *VMDL part*. These programs are written by a VM developer. VMDL's description targets are definitions of the following:

- VM instructions,
- built-in functions, and
- type conversion functions,

which are closely related to JavaScript's logic for handling the eJSVM's internal datatypes. We refer to these as *VMDL functions* for convenience, though VM instructions are not functions. VMDL programs can be regarded as *templates* that cover the operations for all possible combinations of VM internal datatypes. Other aspects of VM implementation, such as VM initialization, GC and so on are written in C.

Because a target JavaScript application needs only a subset of VMDL functions, not all VMDL functions are necessary for generating a customized eJSVM. In the eJSTK, the code selector selects only the required VMDL functions according to the target application's specifications. These specifications are typically defined by the application developer, but the developer is generally not familiar with the internals of an eJSVM. The proposed framework thus provides a profiling mechanism and a specification generator to help application developers define specifications easily. First, the target application is executed by a full-set eJSVM with profiling capabilities on a desktop computer. As a result, log files for the VM instructions and built-in functions are generated. Next, the specification generator produces specification files from these log files. By using these utility tools, the application developer can obtain specifications for the target application without going into the details of the eJSVM.

The VMDL compiler, VMDLC, acts as a translator from VMDL programs to C programs. After selecting the required VMDL functions, VMDLC compiles VMDL programs by eliminating unnecessary parts from the templates in VMDL. It eliminates these parts by using the information in the specifications, and it then outputs the minimum necessary C code fragments. To this end, VMDLC performs static VM datatype analysis. Finally, the eJSVM is obtained by compiling and linking the generated C fragments from the VMDL part and the C part with a C compiler.

In the former version of eJSTK reported in Ref. [21], part of the VM code was written in a simple DSL called vmgen. However, vmgen was specialized for only describing VM instructions and was therefore unable to describe definitions for datatype conversion functions and built-in functions. In addition, vmgen did not perform datatype analysis. Although vmgen was capable of generating small pieces of type dispatch code for VM instructions, this capability was unsatisfactory from the viewpoint of removing as much unnecessary code as possible from a VM, because of the lack of datatype analysis.

2.3 Specifications

There are three kinds of specifications:

- *instruction specifications* for the operands of VM instructions,
- *built-in specifications* for the arguments of built-in functions, and
- *type conversion specifications* for the arguments of datatype conversion functions.

As described in the previous subsection, the specification generator can generate the first two kinds. In contrast, the last kind is automatically generated through VMDLC's datatype analysis, whose details will be described in Section 4.

Each specification is given as an ordered collection of the following form:

name (*operand*$_1$, *operand*$_2$, ...) *action*

where *name* is the name of a VMDL function, and *operand$_i$* is the *i*-th operand or argument of that function. Each *operand* specifies either a VM internal datatype listed in the right column of Table 1, the symbol "_" for all datatypes, or the symbol "-" for non-input operands. Finally, *action* is either `accept`, `unspecified`, or `error`. Here, `accept` indicates that the combination of the specified *operand*s is allowed; `unspecified` indicates that the eJSVM's behavior is not determined for the combination; and `error` indicates that the eJSVM signals an error for the combination.

For example, if only the JS Number datatype is allowed for both operands of an add instruction to perform the addition operation, then its instruction specification is written as follows.

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

Because the first operand of the add instruction is a destination, "-" is specified. For the second and third operands, all combinations of VM internal datatypes corresponding to Number (i.e., `Fixnum` and `Flonum`) are specified as `accept`. Other combinations of datatypes are specified as `unspecified`.

3. VMDL

3.1 Introduction to VMDL through Examples

VMDL is a statically typed DSL whose syntax resembles that of the C language, but it was carefully designed for ease of datatype analysis at the granularity of VM internal datatypes. In particular, VMDL provides a special syntax that allows concise, clear descriptions of type dispatches on the basis of VM internal datatypes. **Figure 2** shows a simplified BNF of VMDL. In this

```
         v ∈ variable name          f ∈ function name          l ∈ label
         t ∈ type name (VM internal datatype, union type, C type)
    ⟨toplevel⟩ ::= ⟨union-def⟩∗ ⟨func-meta⟩∗ ⟨func-def⟩∗
   ⟨union-def⟩ ::= union t = t1 || t2 || ...
   ⟨func-meta⟩ ::= ⟨annotations⟩? f : ⟨func-type⟩
 ⟨annotations⟩ ::= (⟨annotation⟩, ⟨annotation⟩, ...)
  ⟨annotation⟩ ::= vmInstruction | triggerGC | makeInline | ...
   ⟨func-type⟩ ::= t_in -> t_out | (t1, t2, ...) -> t_out
    ⟨func-def⟩ ::= ⟨func-meta⟩ f (v1, v2, ...) ⟨block⟩
   ⟨statement⟩ ::= ⟨block⟩ | ⟨declaration⟩ | ⟨match⟩ | ⟨return⟩
                |  ⟨assignment⟩ | ⟨if⟩ | ⟨do⟩
       ⟨block⟩ ::= { ⟨statement⟩∗ }
 ⟨declaration⟩ ::= t v (= ⟨expr⟩)?;
       ⟨match⟩ ::= l : match (v1, v2, ...) { ⟨case⟩+ }
        ⟨case⟩ ::= case ( ⟨pattern⟩ ) ⟨case-body⟩
     ⟨pattern⟩ ::= t v | !⟨pattern⟩ | (⟨pattern⟩) | ⟨pattern⟩ && ⟨pattern⟩
                |  ⟨pattern⟩ || ⟨pattern⟩ | true
   ⟨case-body⟩ ::= { (⟨statement⟩ | ⟨rematch⟩)∗ }
     ⟨rematch⟩ ::= rematch l (v1, v2, ...);
      ⟨return⟩ ::= return ⟨expr⟩;
  ⟨assignment⟩ ::= v <- ⟨expr⟩;
          ⟨if⟩ ::= if (⟨expr⟩) ⟨statement⟩ else ⟨statement⟩
       ⟨while⟩ ::= while (⟨expr⟩) ⟨statement⟩
        ⟨expr⟩ ::= expressions including constants, variables,
                   binary expressions, function calls, etc.
```

Fig. 2 Simplified BNF of VMDL.

section, we introduce the flavor of VMDL through specific examples.

3.1.1 VM Instructions

First, as an example of a VM instruction definition, **Fig. 3** shows a simplified add instruction. VMDL source code consists of two parts. The first part declares the names of union types by using the `union` keyword and external functions. The second part is the main body of a VMDL function definition.

VMDL handles the following datatypes.

- The VM internal datatypes listed in Table 1.
- Union types that combine multiple VM internal data-types.
- Certain C datatypes that represent data in the C language. The details will be presented in Section 4.2.

A *union type* represents any of the datatypes that are included in the union type. It becomes available in VMDL code by declaring it as shown in Lines 1 and 2. Line 1 defines the union type `Number`, which represents either `Fixnum` or `Flonum`. Line 2 defines the union type `ffs` in the same manner. VMDL offers `JSValue` as a predefined union type for all VM internal datatypes, so the VM developer can thus use `JSValue` without defining it explicitly.

External functions are those defined outside this VMDL definition. In Fig. 3, these are declared in Lines 4–9. It is not required to distinguish whether an external function is written as a C function or as a VMDL function, because even if it is defined in VMDL, it is still compiled to a C function by VMDLC. An external function is declared by giving its name followed by "`:`", and then the datatypes of its arguments and return value in the following format.

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

Fig. 3 Simplified version of the add instruction.

argumentTypes -> *returnType*

When there are multiple arguments, their datatypes before "->" must be separated by commas and enclosed in parentheses.

An external function can have *annotations* attached in parentheses immediately before the function name. Every annotation gives a *characteristic* of the external function, such as "it may trigger GC" and "it may be called from the C part". For example, `int_to_number` is annotated with `triggerGC` (Line 5), which indicates that this function may trigger GC. Given this annotation, VMDLC automatically inserts push/pop operations as described in Section 1 for GC target values, if any, before and after the call to `int_to_number`.

The definition of the VMDL `add` function begins at Line 11. VMDL functions can be annotated in the same way as external functions, and two annotations, `vmInstruction` and `triggerGC`, are specified here. The `vmInstruction` indicates that this function defines a VM instruction operation. A VMDL function annotated with `vmInstruction` is compiled to a C code fragment of the threaded interpreter function [2]. The code of a `vmInstruction` function can use special built-in variables for accessing the VM's internal state, including the VM's registers and runtime stack. Next, the datatypes of the arguments and return value come immediately after the function name, in the same form as for external functions. Here, the arguments are the input operands for this instruction, and the return value is the output operand. In the case of the `add` instruction, the first operand is the output register, and the second and third operands are the input registers. Thus, the two arguments `v1` and `v2` represent the second and third operands, and the return value represents the first operand.

The main body of the instruction definition begins at Line 13. It consists of a `match` statement labeled with `top`. This statement describes a type dispatch by the VM internal datatypes of `v1` and `v2`. The first `case` clause (Line 16) is for the case when both operands are of the `Fixnum` type. The second `case` clause (Line 21) is for the case when both operands are of the `String` type. The third `case` clause (Line 25) is for the case when `v1` is of the `String` type and `v2` is of the `ffs` type, i.e., `Fixnum`, `Flonum`, or `Special`. The last `case` clause (Line 31) has the condition `true` and is for cases not matching any of the above conditions. In the third `case` clause, the `to_string` function converts the datatype of `v2` to `String`. This function, whose definition will be given later, is declared as an external function in Line 7. The result of the datatype conversion is stored in the local variable `s`. Local variables must be declared with their datatypes, and in this case, the datatype of `s` is `String`. After the type conversion, type dispatch must be performed again for `v1` and the conversion result `s`. This is done by the `rematch` statement in Line 29; `rematch` specifies the label (in this example, `top`) of the `match` to be dispatched again and the values to be used in the `match`.

3.1.2 Datatype Conversion Functions

Next, as an example of a datatype conversion function, **Fig. 4** shows a simplified version of the `to_string` function.

Datatype conversion functions from an individual type to `String`, such as `fixnum_to_string`, are declared as external functions in Lines 3–5. Annotations given to `to_string` are

```
 1  union ffss = Fixnum || Flonum || String || Special
 2
 3  fixnum_to_string : Fixnum -> String
 4  flonum_to_string : Flonum -> String
 5  special_to_string : Special -> String
 6
 7  (makeInline) to_string : ffss -> String
 8  to_string (v) {
 9    top: 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  }
```

Fig. 4 Simplified `to_string` function.

specified in Line 7. Here, `makeInline` indicates that this function is subject to the inlining optimization, which will be described in Section 4.5. The function definition begins at Line 8. The datatypes for a function's arguments specify those of the actual arguments. In this example, `to_string` accepts a single argument whose datatype is `JSValue` (i.e., any VM internal datatype), and it returns a `String` value. The main body of the definition is Lines 8–23. This function is defined by using a `match` statement to perform type dispatch according to the datatype of `v` and then returning the result of the appropriate type conversion function for each case.

3.1.3 Built-In Functions

As an example of defining a built-in function, **Fig. 5** shows the `Array.prototype.every` method. Roughly speaking, this method returns whether every element in a receiver array satisfies a given callback function or not. In the VMDL code, this method is implemented as a VMDL function `array_every`, whose first argument (`a`) is the receiver array and second argument (`fn`) is the callback function. The `every` method can take an optional argument that specifies the receiver of every call to the callback function; if it is not given, `undefined` is used.

Because the callback function must be a user-defined function (`Function`) or a built-in function (`Builtin`), the datatype of the callback function argument is of the union type `Callable` (Line 2). Lines 4–15 list the types for auxiliary functions. Line 17 gives the annotations for `array_every`. The `builtinFunction` annotation states that the VM function is a built-in function. The eJSVM internally manages a *context* for the current execution state, including the runtime stack and VM registers. Certain VMDL functions and external functions in VM code require the context to be passed. For such functions, the VM developer can use the `needContext` annotation. When generating C code, VMDLC automatically provides a context argument to every function annotated with `needContext`.

Lines 19–44 give the definition body. In the body of a VM function annotated with `builtinFunction`, a pseudo-variable `na` is available and has the number of actual arguments. The body uses a `while` statement to iterate on the elements of the receiver array. In the body of the `while` statement, a `match` statement

```
union Number = Fixnum || Flonum
union Callable = Function || Builtin

to_boolean : JSValue -> Special
number_to_int : Number -> int
(triggerGC) int_to_number : int -> Number
get_jsarray_length : Array -> Number
(triggerGC) get_array_element : (Array, int) -> JSValue
(triggerGC) set_array_element :
(Array, int, JSValue) -> void
has_array_element: (JSValue, int) -> int
(triggerGC) send_function3 :
  (JSValue,Function,JSValue,JSValue,JSValue) -> JSValue
(triggerGC) send_builtin3 :
  (JSValue,Builtin,JSValue,JSValue,JSValue) -> JSValue

(builtinFunction, needContext, triggerGC)
array_every : (Array, Callable, JSValue) -> JSValue
array_every (a, fn, this) {
  int len = number_to_int(get_jsarray_length(o));
  JSValue t = na >= 2 ? this : JS_UNDEFINED;
  int k = 0;
  while (k < len) {
    if (has_array_element(a, k)) {
      Number jsk = int_to_number(k);
      JSValue val = get_array_element(a, k);
      JSValue result;
      match (fn) {
        case (Function fn) {
          result <- send_function3(t, fn, val, jsk, a);
        }
        case (Builtin fn) {
          result <- send_builtin3(t, fn, val, jsk, a);
        }
      }
      if (result == JS_FALSE || (result != JS_TRUE &&
             to_boolean(result) == JS_FALSE)) {
        return JS_FALSE;
      }
    }
    k <- k + 1;
  }
  return JS_TRUE;
}
```

Fig. 5 Simplified every function.

performs type dispatch on the basis of the VM internal datatype of fn. The callback function is called by send_function3 or send_builtin3.

3.2 Description Based on VM Internal Datatypes

In VMDL, VM internal datatypes can be used as variable types, VMDL function arguments, and return values. In addition, union types that combine multiple VM internal datatypes can be used. These enable datatype analysis at the granularity of VM internal datatypes according to the control flow of VMDL code. At every position in the VMDL code, this analysis finds the set of VM internal datatypes that can be stored for each variable in its scope at runtime. The details of the analysis will be described in Section 4.1.

3.3 Variables

There are two kinds of variables in VMDL:

- variables that hold VM internal datatype values (*VM variables*), and
- variables that hold C language datatype values (*C variables*).

Both kinds of variables must be declared in advance with their datatypes. A VM variable obeys the single assignment rule, i.e., only one assignment to it is allowed on every possible execution path. This simplifies the datatype analysis by VMDLC. Note that assignments to the same variable on different paths are allowed, unlike with static single assignment (SSA) form. The declared type of a VM variable works as a *static assertion*. Thus, the value to be assigned to a VM variable should be consistent with the declared datatype. In other words, a VM variable declared as a union type can be assigned a value of any of the VM datatypes in the union type. For example, a Fixnum value can be assigned to a variable declared as JSValue, but it is a static error to assign a String value to a variable declared as Fixnum.

In contrast, VMDL does not allow union types for C datatypes. Thus, a C variable is allowed to be reassigned. Expressions assigned to C variables and function arguments must match the declared C datatypes.

3.4 Function Annotations

Annotations can be attached to VMDL functions and external functions. There are three kinds of annotations: those that describe the properties of the target function, those that specify how VMDLC handles the VMDL function, and those that are specific to eJSVM. In the examples given in Section 3.1, triggerGC is an instance of the first kind of annotation, while makeInline is an instance of the second kind. An instance of the third kind is builtinFunction. The first and second kinds of annotations are language-common and can therefore be extended to other target languages.

A VMDL function calling a triggerGC function must also be annotated with triggerGC.

3.5 Interfaces with the C Language

In addition to the VM internal datatypes, VMDL defines *C datatypes*, which represent types in C. For example, the int and double types are the same as those in C, cstring represents a C string, and CValue represents a black-box value whose content cannot be manipulated from the VMDL code. It is not permitted to define a union type that includes C datatypes. For arithmetic and relational operations in VMDL, the same arithmetic conversions are applied as in C.

VMDL allows the VM developer to define types that are mapped to structures and arrays. By using such mapped types, whose details are omitted in this paper, a VMDL program can access the members of composite types.

4. VMDL Compiler

The VMDL compiler, VMDLC, converts VMDL code to C code. Given the VM internal datatypes required by the target application as specifications, VMDLC analyzes all the VMDL code and outputs C source code for a customized VM with the minimum required features for the target application.

4.1 Datatype Analysis

We informally explain the datatype analysis by VMDLC by using a specific example.

VMDLC performs static type analysis on a VMDL code by calculating the *type environment* at every point in the code from the top to the bottom direction. Here, a type environment is a set of mappings between variables and their possible datatypes. Note that every VM variable in a type environment is mapped to a *sin-*

Table 2 Calculated type environments at each point in Fig. 3.

	1st round	2nd round
(1)	{{v1 ↦ Fixnum, v2 ↦ Fixnum}, {v1 ↦ String, v2 ↦ Fixnum}}	{{v1 ↦ Fixnum, v2 ↦ Fixnum}, {v1 ↦ String, v2 ↦ Fixnum}, {v1 ↦ String, v2 ↦ String}}
(2)	{{v1 ↦ Fixnum, v2 ↦ Fixnum}}	same
(3)	{}	{{v1 ↦ String, v2 ↦ String}}
(4)	{{v1 ↦ String, v2 ↦ Fixnum}}	same
(5)	{{v1 ↦ String, v2 ↦ Fixnum, s ↦ String}}	same
(6)	{}	same

gle VM internal datatype. This means that type environments do not use union types; instead, if a VM variable has a union type, it is represented as a set of types environments. For example, if v's datatype is Number, then the mapping {{v ↦ Fixnum}, {v ↦ Flonum}} is used instead of {v ↦ Number}.

One drawback of this representation is that the number of type environments in a set can grow exponentially. As a result, unacceptably large amount of computation time and space might be required by VMDLC in the worst case. However, this undesirable situation did not occur with respect to building customized eJSVMs for benchmark programs used in the evaluation in Section 6, because the number of VM variables of union types declared in the same scope was small. In fact, all VMDLC compilation times of VMDL functions were within 500 milliseconds.

In the following description, we simply write "type environment" to refer to a "set of type environments". The type environment at the entrance of a VMDL function is determined by the specification given by the application developer.

Type environment calculation proceeds along the control flow of the target VMDL program. At a join point of two flows, e.g., at the bottom of an if statement, we take the union of the two calculated type environments. For a match statement, the type environment calculation may be iterated until the environment does not change, because internal rematch statements alter the control at the beginning of the match statement. Calculated type environments are used to eliminate unnecessary code that is never executed by the target application.

As an example of a datatype analysis process, consider the add instruction shown in Fig. 3 when the following instruction specification is given.

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

Because rematch is used in the VMDL add function, the type environment calculation is repeated twice. **Table 2** lists the calculated type environment at each comment position in Fig. 3.

In the first round, the type environment at position (1) is {{v1 ↦ Fixnum, v2 ↦ Fixnum}, {v1 ↦ String, v2 ↦ Fixnum}}, according to the given instruction specification. In the type dispatch by the match statement, only the type conditions of the first and third case clauses can be selected. The type environments at the top of the first case clause (Line 17, (2)) is {{v1 ↦ Fixnum, v2 ↦ Fixnum}}, and that at the top of the third case clause (Line 26, (4)) is {{v1 ↦ String, v2 ↦ Fixnum}}. These are obtained by restricting v1's datatype according to the previous type environment at (1) through the case conditions. Because the datatype conditions of the second and fourth case clauses cannot match the type environment at (1), the type environments at (3) and (6) are both empty.

Inside the third case clause (Line 25), the local variable s is declared, and the initialization expression has a String value. The type environment at (5) thus becomes {{v1 ↦ String, v2 ↦ Fixnum, s ↦ String}}. Here, the rematch statement assigns v1 to the variable v1 and s to the variable v2, and a type dispatch is performed again with the match statement labeled with top (Line 14). Hence, the second round of type analysis begins with the type environment at (1) extended to {{v1 ↦ Fixnum, v2 ↦ Fixnum}, {v1 ↦ String, v2 ↦ Fixnum}, {v1 ↦ String, v2 ↦ String}}.

The type environment at each point in the second round is listed in the right column of Table 2. In this round, the second case clause (Line 21) of add is executed, because the type environment at (1) has been extended, and the type environment at (3) becomes {v1 ↦ String, v2 ↦ String}. The second case clause has no rematch statement, and the other case clauses do not change the type environment at (1) any further. The type environment calculation thus terminates. The right column of Table 2 lists the final type environments for all points in the program.

As seen in the table, the type environment at the top of the fourth case clause (Line 32, (6)) is empty, which implies that the clause is never executed. Accordingly, VMDLC eliminates such clauses with the empty type environments to generate the minimum necessary code according to the instruction specification.

4.2 Datatypes of Variables

As described in Section 3.3, a VM datatype variable declared as a union type can be assigned a value of any VM datatype in that union type. In the datatype analysis process, the datatypes for such variables in type environments are not the declared types but the types of the actually assigned values. For example, consider the following VMDL code fragment.

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

The variable v is declared as JSValue, but a String value is assigned at (1), and a Flonum type value is assigned at (2). In both execution paths of the if statement, a value is assigned to v, so this code has no "unassigned to variable" error. The type environment at (1) is $\{\{\texttt{x} \mapsto \texttt{JSValue}, \texttt{v} \mapsto \texttt{String}\}\}$, and that at (2) is $\{\{\texttt{x} \mapsto \texttt{JSValue}, \texttt{v} \mapsto \texttt{Flonum}\}\}$. As a result, the type environment at (3) becomes $\{\{\texttt{x} \mapsto \texttt{JSValue}, \texttt{v} \mapsto \texttt{String}\}, \{\texttt{x} \mapsto \texttt{JSValue}, \texttt{v} \mapsto \texttt{Flonum}\}\}$. Thus, if we write code that assumes that the value of v is a String at (3), VMDLC will report an error. An example of such error code is f(v), for which f's argument is restricted to a String value.

Note that VMDLC performs the type analysis independently for VM datatype variables and C variables,

4.3 Annotation Processing

When compiling a function annotated with vmInstruction or builtinFunction, VMDLC automatically inserts specific code fragments to prepare special built-in variables, e.g., those for accessing the interpreter's internal state in the case of vmInstruction.

When a VMDL function annotated with triggerGC is called, VMDLC automatically inserts push/pop operations for VM variables to/from the GC root. In some cases, these operations do not have to be inserted, as will be described in Section 4.5.3.

Because of the automatic code insertion by VMDLC, the VM developer only needs to pay attention to the essential program logic of VMDL functions, without having to deal with the details mentioned above.

4.4 Generation of Type Conversion Specifications

Datatype conversion functions are mainly called from VM instructions and built-in functions. Their datatypes are determined by the instruction specifications and built-in specifications, respectively. By using the information in these specifications, type conversion specifications can be automatically obtained without the application developer having to deal with them. Thus, VMDLC has a mechanism for automatically generating type conversion specifications. Through this mechanism, the proposed framework performs inter-VMDL function datatype analysis to produce the minimum required VM code in C.

As a simple example, we consider the VM add instruction shown in Fig. 3 and the to_string datatype conversion function shown in Fig. 4. Suppose that we have the following instruction specification for add.

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

For simplicity, suppose also that to_string is called only from the add instruction. Because v2's datatype in the type environment at Line 27 in Fig. 3 is Fixnum or Flonum, VMDLC outputs the following specification of to_string.

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

In this way, VMDLC obtains the datatype information for the necessary arguments of every datatype conversion function.

If a datatype conversion function is called from the C part of the VM source code, VMDLC stops computing the necessary datatype information and instead outputs the specification as "all datatypes are accepted". Such a function is annotated by the VM developer with a calledFromC annotation.

4.5 Optimizations

VMDLC performs multiple optimizations by using the results of the datatype analysis. Note that it does not perform the general optimizations performed by a C compiler, because the generated C code is compiled separately by the C compiler.

4.5.1 Function Inlining

When a VMDL function is called in a case branch of a match statement, it may perform a type dispatch again, even though a type dispatch has already been performed in the match statement. Consider the add function shown in Fig. 3 again. If the case clause at Line 25 is selected, the to_string function shown in Fig. 4 is called at Line 27. In to_string, a type dispatch is performed again according to its argument, which comes from v2 in add. As a result, duplicated type dispatches on the same value occur in both the caller and the callee of to_string. To avoid this, the *function inlining* optimization eliminates such redundant, useless type dispatches.

To enable inlining, the makeInline annotation is given to the function to be inlined. For the call to the to_string function in Line 27 of the add function, the argument v2 is already known to be of Fixnum type because of the type environment at (4). Thus, this call is replaced with a call to the fixnum_to_string function, which is returned in the case clause for Fixnum at Line 14 in Fig. 4. As described above, the function inlining here is designed to optimize type dispatches. Unlike general inlining, function inlining is performed only when the replacement consists of a single expression such as fixnum_to_string(v2). This is because the purpose of this inlining optimization is to eliminate redundant type dispatches while preventing the VM size from growing at the same time.

Function inlining has another advantage besides the elimination of redundant type dispatches. The VM here has two kinds of type conversion functions: those that accept *all* VM internal datatypes for an argument, such as the to_string function, and those that expect a *unique* VM internal datatype, such as the fixnum_to_string function. We refer to the former as $\texttt{to\_}y$ *functions* and to the latter as $x\texttt{\_to\_}y$ *functions*. The inlining optimization enables the VM developer to use $\texttt{to\_}y$ functions even in a context in which the source datatype of a conversion is uniquely determined without causing extra overheads for duplicated type dispatches. This capability enhances the descriptiveness and reduces the burden on the VM developer who does not need to be fully aware of type environments or to worry about identifying which $x\texttt{\_to\_}y$ function to use.

4.5.2 Type Condition Splitting

The function inlining optimization cannot be applied in a context in which the argument datatype of a function call to be inlined is not uniquely determined. For example, suppose that the following instruction specification is given to the add function shown in Fig. 3.

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

```
 1  (vmInstruction, triggerGC)
 2  add : (JSValue, JSValue) -> JSValue
 3  add (v1, v2) {
 4    top: match (v1, v2) {
 5      ...
 6      case (String v1 && Fixnum v2) {
 7        // {{v1 ↦ String, v2 ↦ Fixnum}}
 8        String s = to_string(v2);
 9        rematch top(v1, s);
10      }
11      case (String v1 && Flonum v2) {
12        // {{v1 ↦ String, v2 ↦ Flonum}}
13        String s = to_string(v2);
14        rematch top(v1, s);
15      }
16      case (String v1 && Special v2) {
17        // {}
18        String s = to_string(v2);
19        rematch top(v1, s);
20      }
21      ...
22    }
23  }
```

Fig. 6 A `case` clause with splitting type conditions.

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

The only difference from the previous instruction specification given in Section 4.1 is that the combination of `String` and `Flonum` is also allowed, in the third line. With this instruction specification, the type environment at (4) in the `add` function becomes $\{\{\texttt{v1} \mapsto \texttt{String}, \texttt{v2} \mapsto \texttt{Fixnum}\}, \{\texttt{v1} \mapsto \texttt{String}, \texttt{v2} \mapsto \texttt{Flonum}\}\}$, where `v2` has two possible datatypes, `Fixnum` and `Flonum`. Here, function inlining is not permitted for `to_string(v2)` in Line 27, because `v2`'s type is not uniquely determined. There are two possible replacements; the first is `fixnum_to_string(v2)`, and the second is `flonum_to_string(v2)`. This situation occurs because the `case` clause specifies the type condition by using the union type `ffs` for the variable `v2`.

For such a case, VMDLC enables function inlining by splitting the `case` clause so that the condition for a union type is divided into conditions for the individual datatype. By splitting the `case` clauses in the above example, the `add` function would appear as shown in **Fig. 6**, in which the `case` clause of the original code (Line 25 in Fig. 3) is divided into three `case` clauses (Lines 6, 11, and 16 in Fig. 6). Because `v2`'s datatype is now unique in each `case` clause, the inlining optimization can be applied. Specifically, `to_string(v2)` in Line 8 is replaced with `fixnum_to_string(v2)` and `to_string(v2)` in Line 13 is replaced with `flonum_to_string(v2)`. Note that the `case` clause starting at Line 16 will be eliminated and not included in the C code, because the type environment is empty.

Note, however, that splitting a `case` clause increases the number of `case` clauses and duplicates the body of the original `case` clause in the VMDL code, which might increase the VM size. Accordingly, it is necessary to limit the type conditions to be split to those that are expected to be effective. To this end, VMDLC accepts options from the application developer to narrow down the type conditions to be split and performs splitting only when function inlining becomes available.

4.5.3 Push and Pop Elimination

As another optimization, VMDL suppresses the generation of unnecessary push/pop operations in the following four cases.

First, in an eJSVM, some VM internal datatypes are represented as immediate values without any pointers. For example, a `Fixnum` value has a two's complement representation embedded in a word. Even if GC occurs, it is not necessary to perform push/pop operations for such immediate values because these contain no pointers. As a result of VMDLC's type analysis, push/pop operations can be eliminated when the type environment indicates that a VM variable is of a datatype with an immediate representation, like `Fixnum`.

Second, some functions annotated with `triggerGC` cause GC only when specific datatypes are passed. For example, consider the `to_double` function, which is given a value of a VM internal datatype and converts it to a C `double` value. This function causes GC only when a value of JS datatype `Object` (e.g., `Simple_object` and `Array`) is given, because it allocates intermediate data on the heap, but it does not do so when other datatypes are given. Accordingly, if type analysis indicates that the actual argument of a `to_double` call is of a VM internal datatype that does not cause GC, and if this call has been suitably replaced through inlining, then it is not necessary to wrap the replaced call with a push/pop.

Third, push/pop operations are not inserted for a VM variable that is never used after a call of a `triggerGC` function.

Fourth, when multiple `triggerGC` functions are successively called, pop and push operations for the same variables between the two such function calls are eliminated. For example, for a series of two `triggerGC` VM function calls "`f(); g();`", and two `Value` variables, a and b, VMDLC generates

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

instead of the following redundant code.

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

For simplicity, VMDLC applies this optimization within every *basic block*.

4.6 Generation of C Code

VMDLC generates C code from VMDL code unless it detects any errors via its datatype analysis. Each corresponding C statement is uniquely and naturally determined from a VMDL statement except for the `match` and `rematch` statements. For `match` and `rematch`, VMDLC outputs (nested) `switch` and `goto` statements by using the algorithm given in our previous paper [21].

4.6.1 Example of VM Instruction

Figure 7 shows the formatted C code generated from the VM `add` instruction shown in Fig. 3, given the instruction specification described in Section 4.4. Although the specification does not specify the case of `add (-,String,String)` as accepted, the VMDL's datatype analysis judges this case as necessary, because the `rematch` statement in Line 29 in the `add` instruction requests the case in which both operands are `String` values.

This code fragment is included in the main loop of the threaded interpreter. In that main loop, `v1` and `v2` have been appropriately defined to denote the first and second input operands, respectively. Line 1 generates a label for the `add` instruction. The `match` statement in the VMDL code is compiled to nested `switch` statements. The eJSVM uses the tag values described in Section 1 to identify VM internal datatypes, and these are obtained

```
 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:
```

Fig. 7 Generated C code for the VM add instruction.

by using `get_ptag` (Lines 4 and 6). The outer `switch` in the generated code first dispatches according to `v1`'s datatype, and in the `String` case (Line 5), the inner `switch` dispatches according to `v2`'s datatype. Note that, in the `Fixnum` case (Line 22) for the outer branch, an inner `switch` statement for checking whether `v2` is a `Fixnum` is unnecessary. This is because `add (-,Fixnum,Fixnum)` is the only acceptable combination of datatypes in which the first operand is a `Fixnum`, while the other combinations are `unspecified`; we can thus assume that `v2` is of the `Fixnum` type if the first operand is of a `Fixnum`. The assignments to `regbase[r0]` (Lines 9 and 26) set the results to the destination of the `add` instruction. For the `rematch` statement, the generated code reassigns both `v1` and `v2` appropriately (Line 18), and then goes to the top of the outer `switch`. Although redundant assignments are generated, we expect the C compiler to remove such redundancies.

4.6.2 Example of Built-In Function

Figure 8 shows the formatted C code generated from the VM `array_every` function shown in Fig. 5, given the built-in specification of "`array_every (_,_,_) accept`".

Because the VMDL function has been annotated with `needContext`, the context argument is automatically provided. In addition, from the `builtinFunction` annotation, `fp` (frame pointer) and `na` (number of arguments) are also added as arguments. In an eJSVM implementation, the arguments given to a built-in VMDL function (e.g., `a`, `fn`, and `this` for `array_every`) are given via an array of VM registers, which is named `args`. Line 2 is added by the `builtinFunction` annotation and makes `args` available for accessing actual arguments. Lines 3–5 obtain the receiver, the callback function, and the `this` object from `args` and assign them to their corresponding local `Value` variables. Because `int_to_number` and `get_array_element` have been annotated with `triggerGC`, `push`/`pop` operations

```
 1  void array_every (Context* context, cint fp, cint na) {
 2    builtin_prologue();
 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        PUSH(&t, &fn, &a);
12        Value jsk = int_to_number(k);
13        PUSH(&jsk);
14        Value val = get_array_element(a, k);
15        POP4();
16        Value result;
17        switch ((unsigned int) get_htag(fn).v) {
18        case HTAGV_BUILTIN:
19        default:
20          {
21            PUSH5(&a, &t, &fn, &val, &jsk);
22            result = send_builtin3(t, fn, val, jsk, a);
23            POP5();
24          }
25          break;
26        case HTAGV_FUNCTION:
27          {
28            PUSH5(&a, &t, &fn, &val, &jsk);
29            result = send_function3(t, fn, val, jsk, a);
30            POP5();
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  }
```

Fig. 8 Generated C code for the VM `array_every` function.

are automatically inserted. Note that because of the `push`/`pop` elimination optimization, redundant `pop`s and `push`es have been removed between these two function calls (Lines 12–14). Here, `PUSH`*n* and `POP`*n* are macros that are expanded to *n* `push`es and *n* `pop`s, respectively.

The `switch` statement (Line 6) generated from the `match` statement in the VMDL code performs a type dispatch by using the "header tag" value in the object body of a `Function` or `Builtin`, which is obtained by using `get_htag`. In the eJSVM, the return value of a built-in function has to be stored in a special register by using `set_a` (Lines 36 and 42).

5. Utility Tools

5.1 Profiler and Specification Generator

To help the application developer prepare instruction specifications and built-in specifications, the proposed framework provides the *profiler* and the *specification generator*. The profiler is a full-set eJSVM that can record logs of the names and datatypes of the operands/arguments of executed VM instructions and built-in functions. Note that logs are not recorded for type conversion functions, because their specifications can be inferred and generated by VMDLC. The profiling is meant to be performed on a desktop computer, with execution of the target application by the full-set eJSVM. Specifications are obtained by giving the generated log files to the specification generator.

When using this profiler, the application developer is expected to iterate profile runs by giving various inputs until all possible datatypes that the application might use are obtained. For an application that is given a fixed input, such as one in the benchmark programs used in Section 6, its minimum specifications can be obtained with a single profile run.

5.2 Code Selector

In general, there are VM instructions and built-in functions that are never used by the target application. The specification files generated by the specification generator contain only `unspecified` lines for these cases. For example, if the target application does not use the VM `leftshift` instruction, the following line is generated for this instruction in the instruction specification file.

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

The *code selector* excludes the VMDL files for `unspecified` VMDL functions and leaves only the files required for later processing by VMDLC. Note that the code selector excludes both unnecessary VM instructions and unnecessary built-in functions.

6. Evaluation

6.1 Evaluation Setup

To evaluate the proposed framework, we generated eJSVMs for the following two devices:

- a Raspberry Pi 3 Model B+ (RP), and
- a FRDM-K64F (FD).

We used the latter as an instance of small devices for embedded systems. **Table 3** summarizes the details of the execution environments for those devices. For both environments, we generated eJSVMs with a 32-bit configuration. By using the profiler and the specification generator, we prepared the *minimum* specifications for each benchmark program that we used in the evaluation. Of course, the minimum specifications may differ from program to program. Each eJSVM generated from these specifications was an eJSVM customized for each benchmark program.

We used eight programs from the AreWeFastYet benchmark [15] and 12 programs from the SunSpider benchmark [*2]. These programs were slightly modified to run on an eJSVM. In addition, we used the dht11 benchmark program [17], which we created on the basis of a program that was actually used in IoT devices. It repeatedly converts a sequence of bits from a temperature and humidity sensor into numerical temperature and humidity values.

For each benchmark program, we used the following eJSVMs

Table 3 Execution environments for evaluation.

	RP	FD
CPU	Cortex-A53 (ARMv8) 64-bit SoC	Cortex-M4F (ARM)
Frequency	1.40 GHz	120 MHz
Memory	1 GB	256 KB RAM + 1,024 KB flash
OS	Raspbian 9.13	mbed-os-6.14.0
C compiler	GCC 6.3.0 20170516 +deb9u1	GCC 7.3.1 20180622 (15:7-2018-q2-6)

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

in the evaluation.

- An eJSVM of the previous version [21] (Prev), which was generated by the old DSL (vmgen) for describing VM instructions. The old DSL could describe type dispatching conditions, but the code for each branch had to be given by a C code fragment. Thus, we gave up the analysis of the C code on the basis of VM internal datatypes. The previous version was also not equipped with a code selector mechanism to exclude unnecessary function definitions. Note that this mechanism was implemented in VMDLC.
- An eJSVM generated by VMDL without any optimization (Opt^-).
- An eJSVM generated by VMDL with inlining optimization (Opt^I).
- An eJSVM generated by VMDL with both inlining and type condition splitting optimizations (Opt^IS).

When generating eJSVMs on the RP with the C compiler, we enabled inline caching [5] and hidden class [4] caching at allocation sites. When generating eJSVMs on the FD, however, we had to disable these capabilities because of the limited RAM size. All of the VMs used the garbage collection with the Fusuma compaction [18].

6.2 Descriptiveness and Safety

First, we evaluated whether VMDL improved both the descriptiveness and safety of VM programs.

6.2.1 Type Dispatch

In C, type dispatches based on multiple datatypes are described by nested and usually complicated `switch` statements. In contrast, we found that VMDL allowed the VM developer to write type dispatches flatly by using `match` statements. Moreover, datatype conditions could be expressed concisely by using logical operations.

6.2.2 Automatic Push and Pop Insertion

The annotations given to VMDL functions provide information to VMDLC, which enables it to generate a program that follows the internal implementation of the VM. Because of this mechanism, the VM developer can write VMDL functions in a concise manner without having to deal with the VM's issues in detail.

Automatic insertion of push/pop code for a `triggerGC` function is a typical example of how the descriptiveness of VM programs can be improved. In the VM code of the previous eJSVM, denoted as Prev, 36 save and restore operations were contained in the VM instructions, type conversion functions, and so on, and these were manually written in C with the VM developer's careful coding. With the current eJSVM, however, these descriptions were replaced with VMDL code. As a result, manual insertions of these explicit push/pop operations was no longer necessary. This shows the effectiveness of improving the descriptiveness of VM code and reducing the burden on the VM developer. In addition, the automatic push/pop insertion improved the safety of the VM code, because errors from forgetting to insert the necessary code for those operations could no longer occur.

6.2.3 Context Passing

As described in Section 3.1.3, the `needContext` annotation can be attached to functions that require the context to be

Table 4 VM sizes (in bytes) and numbers of inlining optimizations.

Program	VM size (RP)					VM size (FD)					Inlining	
	Prev	Prev$^+$	Opt$^-$	Opt^{I}	Opt^{IS}	Prev	Prev$^+$	Opt$^-$	Opt^{I}	Opt^{IS}	Opt^{I}	Opt^{IS}
AreWeFastYet benchmark												
Bounce	77,876	61,396	60,060	59,964		138,744	101,456	103,440	103,184		22	
List	77,732	61,232	59,804	59,780		138,680	101,440	103,280	103,088		20	
Mandelbrot	78,212	61,772	60,520	60,344	+166	139,216	101,440	104,048	103,912	+312	24	+5 (3)
Permute	77,820	61,324	59,884	59,856		138,752	101,440	103,360	103,160		20	
Queens	77,760	61,264	59,828	59,820		138,688	101,440	103,280	103,088		20	
Richards	78,324	61,828	60,556	60,548		139,432	101,440	104,056	103,824		25	
Sieve	77,792	61,296	59,868	59,860		138,712	101,440	103,328	103,136		20	
Towers	77,912	61,424	59,988	59,972		138,816	101,440	103,424	103,224		20	
SunSpider benchmark												
3d-cube	78,044	61,220	60,472	60,188	+ 20	139,232	107,224	110,048	109,880	+224	33	+4 (2)
access-binary-trees	76,860	59,800	58,208	58,088		138,304	101,104	101,776	101,744		17	
access-fannkuch	76,212	60,100	57,628	57,556		137,480	102,512	101,536	101,504		18	
access-nbody	77,708	60,276	59,172	59,044	+132	139,000	101,624	103,728	103,608	+328	18	+6 (3)
bitops-3bit-bits-in-byte	75,808	58,360	56,548	56,516		136,968	100,720	100,152	100,112		14	
bitops-bits-in-byte	75,736	58,288	56,456	56,436		136,928	100,720	100,112	100,072		12	
bitops-bitwise-and	75,704	58,256	56,472	56,468		136,928	100,720	100,096	100,104		12	
controlflow-recursive	75,852	58,404	56,668	56,600		137,096	100,720	100,288	100,248		14	
math-partial-sums	76,748	59,924	58,448	58,312	+ 68	138,120	111,072	111,880	111,768	+240	23	+4 (2)
math-spectral-norm	77,076	60,228	59,044	58,908	+ 28	138,384	101,624	103,088	102,960	+ 72	22	+2 (1)
string-base64	77,116	61,252	60,088	59,644	+100	138,320	101,864	103,320	102,832	+ 64	33	+1 (1)
string-fasta	77,400	60,912	59,664	59,504	+140	139,176	101,312	104,248	104,152	+ 72	28	+3 (2)
IoT program												
dht11	76,816	60,200	58,864	58,772		137,832	101,040	101,888	101,504		27	

passed. When generating C code for a function annotated with `needContext`, VMDLC automatically supplies the context argument. This annotation frees the VM developer from giving the context argument explicitly, which is necessary when writing VM instructions, datatype conversion functions, and built-in functions directly in C. In this evaluation, we found that all 466 instances of explicit context passing in the Prev code were reduced by using VMDL.

6.2.4 Type Checking

Type checking for VM internal datatypes helps to statically detect type errors. In the case of C, the `Value` type described in Section 1 is the only C datatype that represents all first-class values in the target language, and it is impossible to type check statically on the basis of VM internal datatypes. In contrast, VMDL code can be type checked statically at the granularity of VM internal datatypes, which improves the safety of VM programs. As a specific example, we describe a case study on the `String.prototype.split` method.

The built-in method `split`(*separator*, *limit*) returns an `Array` object in which substrings of the receiver `String` split by *separator* are stored. The second argument, *limit*, specifies a limit on the length of the returned array. With Prev code, we defined this built-in method in C, including the following code.

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

Here, `args[2]` is *limit*, whose datatype is `Value`, and `cint` is a type synonym for C's signed integer. However, this code was a bug, because *limit* was restricted as a `Number`, the expected VM internal datatype for `number_to_cint`'s argument. This is against the specification of `split`, in which any VM internal datatype was allowed for *limit*. This bug was undetected by the C compiler because the C type of the argument was `Value`. In fact, we were not aware of the bug until we implemented the same functionality in VMDL as described below and were notified of the error signaled by VMDLC.

```
 1  union Number = Fixnum || Flonum
 2  number_to_cint : Number -> int
 3  ...
 4  string_split : (String, JSValue, JSValue) -> JSValue
 5  string_split (s, separator, limit) {
 6    ...
 7    int lim = (limit == JS_UNDEFINED)?
 8              ...: number_to_cint(limit);
 9    ...
10  }
```

(a) Incorrect definition.

```
 1  union Number = Fixnum || Flonum
 2  (triggerGC) toInteger : JSValue -> int
 3  ...
 4  string_split : (String, JSValue, JSValue) -> JSValue
 5  string_split (s, separator, limit) {
 6    ...
 7    int lim = (limit == JS_UNDEFINED)?
 8              ...: toInteger(limit);
 9    ...
10  }
```

(b) Correct definition.

Fig. 9 VMDL definitions for `split`.

Next, we defined `split` as the VMDL function `string_split`. First, we used the VMDL code shown in **Fig. 9** (a), which directly corresponded to the above C code. However, VMDLC signaled an error during compilation, because `limit`, the third argument of the built-in function `split`, was declared as a `JSValue` (Line 4), but it was passed as a `number_to_cint` argument declared as a `Number`. These two VM internal datatypes were inconsistent, that is, `JSValue` was not a subset of `Number`. We thus corrected `string_split` in VMDL as shown in Fig. 9 (b). In this code, `toInteger`, whose argument was `JSValue`, was used instead of `number_to_cint`. Through this correction, `string_split` was successfully defined.

6.3 Efficiency

6.3.1 VM Size

Table 4 lists the VM size (in bytes) for each VM generated for the RP and FD. For comparison, we also implemented an extended version of Prev, which we call Prev^{+}, to include the code selection mechanism for built-in functions. In the table, each Opt^{IS} column indicates the difference from the corresponding Opt^{I} column, with an empty cell indicating that there was no difference. The reason for the differences in VM sizes was that the number of `case` clauses was increased by the type condition splitting optimization. As the number of `case` clauses increased, the number of chances to apply inlining optimization also increased. The table also lists the numbers of inlining optimizations, with the numbers of split type conditions in parentheses.

Note that for Prev (and for Prev^{+}), the VM developer manually created the carefully optimized code from the beginning. In contrast, the use of VMDL frees the VM developer from the burden of such careful coding. Nevertheless, the VM sizes for Prev^{+}, Opt^{-}, and Opt^{I} were almost the same, although the sizes of Opt^{-} and Opt^{I} for the RP were a little smaller and those for the FD were a little larger than for Prev^{+}.

Comparing Opt^{-} and Opt^{I}, we can see that inlining optimizations did not affect the VM sizes. This was because we restricted function inlining from a function-call expression to another expression. This suppressed the increase in code size after inlining, and it also eliminated functions that were no longer called from anywhere after inlining.

For every program that applied the type condition splitting optimization, the VM size for Opt^{IS} was larger than that for Opt^{I}, because new `case` clauses were generated. However, the size growth was at most only 166 bytes for the RP and 328 bytes for the FD, more specifically, the growth was about 0.3% of the total size. Despite this, as will be seen in the next subsection, the execution time for Opt^{IS} with such a program was about 5% shorter than that for Opt^{I} on average.

6.3.2 Execution Time

By using VMDL, the VM developer does not have to manually write carefully optimized VM code because of the VMDLC's datatype analysis and various optimizations. In contrast, such careful coding was necessary in the Prev approach. In spite of this, in this evaluation, we found that VMDL could generate VMs that were as efficient as, or even more efficient than VMs generated with Prev code.

Figures 10 and **11** show the execution times, averaged over 50 runs (RP) and 20 runs (FD), of the benchmark programs on the RP and FD, respectively. Here, we only show the results for Opt^{-} and Opt^{I}, while those for Opt^{IS} will be presented later. For each program, the execution times were normalized to the execution time of the Prev case. The geometric mean is also shown. On the FD, Richards and controlflow-recursive failed to run due to lack of memory.

For more than half of the programs, Opt^{-} was slower than Prev. This was because the Prev code was manually optimized by the VM developer in terms of VM internal datatypes, whereas the VMDL code was not. For example, in the Prev code with the old DSL, the VM developer used appropriate x`_to_`y functions in dispatched body code written in C. In contrast, with VMDL, only `to_`y functions were used in `case` clauses, because every `case` clause was the target of type analysis for optimization. However,

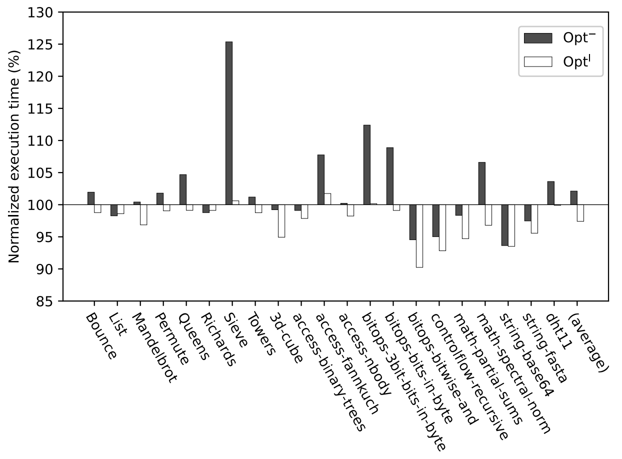


Fig. 10 Normalized execution times (RP).

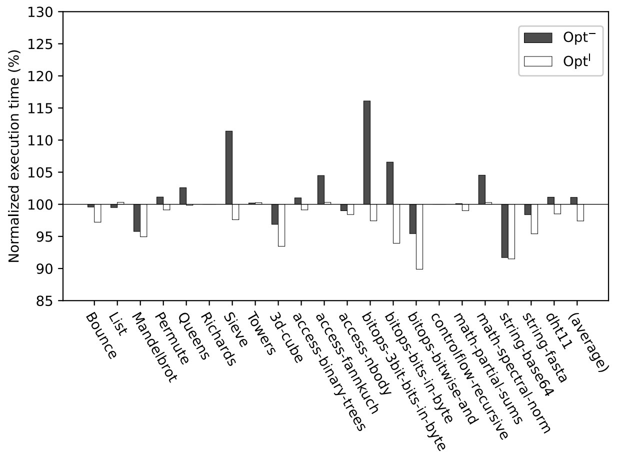
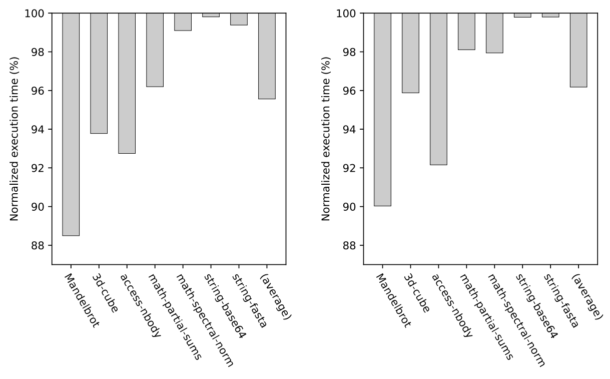


Fig. 11 Normalized execution times (FD).



(a) RP (b) FD

Fig. 12 Normalized execution times for Opt^{IS} as compared to Opt^{I}.

because Opt^{-} did not optimize the code, calls to `to_`y functions remained and caused duplicated type dispatches. Averaged over the results on all the executed benchmarks, the execution times for Opt^{-} were 102.1% (RP) and 101.1% (FD) of those for Prev.

For many of the programs, Opt^{I} was faster than Prev. This indicated that the VMDL code was optimized in terms of VM internal datatypes through function inlining. Averaged over the results on all the executed benchmarks, the execution times for Opt^{I} were 97.4% (RP) and also 97.4% (FD) of those for Prev.

For seven programs that applied the type condition splitting optimization, the execution times for Opt^{IS} were smaller than those for Opt^{I}. **Figure 12** shows the execution time ratios (Opt^{IS}/Opt^{I})

for those programs. Note that this figure shows speedup compared to Opt^{l}, while Figs. 10 and 11 shows the normalized execution times compared to Prev. Both the RP and the FD showed almost the same tendency, with averages of 95.6% and 96.2%, respectively.

7. Related Work

eJS previously used a simple DSL [21], vmgen, to define VM instructions. The syntax of vmgen is limited and focused only on generating dispatch code based on the operand datatypes in the definitions of VM instructions. An instruction is described as a set of operations that are guarded with operand datatypes, and each operation consists of a C code fragment that is directly included in the generated code. The role of the DSL processor is limited to generating a datatype-based dispatch code in terms of C's `match` statement to choose one of the operations. In contrast, VMDL offers a syntax that makes it easy to perform static analysis on the operations of VM instructions, datatype conversion functions, and built-in functions. This enables VMDLC to infer the possible set of datatypes for each VM variable, which enhances error detection and various optimizations.

Latendresse [14] generated a compact VM for the Scheme language by automatically generating new instructions for the VM. That approach assigns new opcodes for repetitive sequences of instructions to reduce the VM size. We also seek to reduce the VM size, but our approach is totally different in that we eliminate unnecessary parts of the VM for the target application.

Jikes RVM [1] is a meta-circular Java VM, that is, a Java VM is written in Java. The Maxine VM [22] is also a meta-circular Java VM. Both of those approaches use a safer general-purpose language than C, namely Java, to describe VM code and thus increase the safety of the VM. In contrast, our approach defines and uses a DSL, called VMDL, that achieves both descriptiveness and safety. Jikes RVM's ahead-of-time (AOT) compiler compiles the class files of the Java VM into native code. Write barriers for GC are woven into the VM executable by the AOT compiler. The AOT compiler also recognizes special annotations, such as one specifying that the annotated method does not cause GC, to control how annotated methods are compiled. VMDL also has similar annotations.

Vmgen [*3] [8], [10] and its successor Tiger [3] are virtual machine interpreter generators with special support for stack machines. Interpreters of VMs such as Gforth [7] and Cacao Java VM [11] were described in Vmgen. Specifically, those approaches defined DSLs and used them to describe instructions, which were essentially C code fragments annotated with meta information such as the effect of an instruction on stack-top elements. The meta-information was used to generate an interpreter with various common optimizations such as stack caching and superinstructions. In contrast to VMDL, Vmgen and Tiger have no special support for dynamically typed values and only C types are used in their DSLs.

There are many works on static analysis for scripting languages and dynamically typed languages [9], [13], [16], [19], [20]. VMDL can be regarded as an instance of a special-purpose scripting language. VMDLC applies a control-flow analysis to calculate type environments.

The Truffle DSL [12] is a language for describing abstract syntax tree (AST) interpreters. It consists of the Java language with special annotations. In Truffle, operations of the target language, such as `add`, are described as AST nodes, and the VM developer implements methods to execute these operations for the given arguments. The Truffle DSL allows the VM developer to implement specialized methods for arguments having specific types. The Truffle framework generates glue code for each AST node that dispatches to specialized methods according to the datatypes of the arguments. An AST interpreter written in the Truffle DSL is executed by a Java VM with a special just-in-time (JIT) compiler [24]. The JIT compiler inlines the execution method of a child node into the execution method of its parent node, which eliminates type-based dispatch code for the child node. As proposed here, the type-based dispatch and inlining of VMDL have essentially the same effects; however, VMDL is meant for embedded systems with restricted resources. Accordingly, the inlining of VMDL is carefully designed not to bloat the VM footprint. VMDL also allows the VM developer to limit possible datatypes by specifying operands.

8. Conclusion

This paper proposes a DSL for describing VM programs for embedded systems. This DSL enables static VM datatype analysis and compilation into C programs by applying type-based optimizations. The proposed approach is implemented as a framework for developing customized eJSVMs, JavaScript VMs for embedded systems, which are specialized for specific target applications. The framework consists of VMDL (the DSL), VMDLC (a VMDL compiler), and related utilities. VMDL enables description of VM programs on the basis of VM internal datatypes, and VMDLC analyzes the programs at the granularity of these internal datatypes, which is difficult for VM programs that are directly written in C. The proposed approach largely solves the descriptiveness, safety, and VM size problems described in Section 1.

Acknowledgments This work was partly supported by JSPS KAKENHI Grant Number JP18KK0315.

References

[1] Alpern, B., Augart, S., Blackburn, S.M., Butrico, M.A., Cocchi, A., Cheng, P., Dolby, J., Fink, S.J., Grove, D., Hind, M., McKinley, K.S., Mergen, M.F., Moss, J.E.B., Ngo, T.A., Sarkar, V. and Trapp, M.: The Jikes Research Virtual Machine project: Building an open-source research community, *IBM Syst. J.*, Vol.44, No.2, pp.399–418 (2005).

[2] Bell, J.R.: Threaded Code, *Comm. ACM*, Vol.16, No.6, pp.370–372 (1973).

[3] Casey, K., Gregg, D. and Ertl, M.A.: Tiger – An Interpreter Generation Tool, *Proc. Compiler Construction, 14th International Conference* (*CC 2005*), Lecture Notes in Computer Science, Vol.3443, pp.246–249, Springer (2005).

[4] Chambers, C., Ungar, D.M. and Lee, E.: An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes, *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications* (*OOPSLA 1989*), pp.49–70, ACM (1989).

[5] Deutsch, L.P. and Schiffman, A.M.: Efficient Implementation of the Smalltalk-80 System, *Proc. 11th Annual ACM Symposium on Principles of Programming Languages* (*POPL 1984*), pp.297–302, ACM (1984).

[*3] Note that this is a different technology from vmgen in the previous version of eJSTK.

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

[7] Ertl, M.A.: A Portable Forth Engine, *Proc. EuroFORTH '93 Conference* (1993).

[8] Ertl, M.A., Gregg, D., Krall, A. and Paysan, B.: Vmgen: A generator of efficient virtual machine interpreters, *Softw., Pract. Exper.*, Vol.32, No.3, pp.265–294 (2002).

[9] Graver, J.O. and Johnson, R.E.: A Type System for Smalltalk, *Conference Record of 17th Annual ACM Symposium on Principles of Programming Languages* (*POPL 1990*), pp.136–150, ACM (1990).

[10] Gregg, D. and Ertl, M.A.: A Language and Tool for Generating Efficient Virtual Machine Interpreters, *Proc. Domain-Specific Program Generation*, pp.196–215 (2003).

[11] Gregg, D., Ertl, M.A. and Krall, A.: Implementing an Efficient Java Interpreter, *Proc. High-Performance Computing and Networking, 9th International Conference*, *HPCN Europe 2001*, pp.613–620 (2001).

[12] Humer, C., Wimmer, C., Wirth, C., Wöß, A. and Würthinger, T.: A domain-specific language for building self-optimizing AST interpreters, *Proc. 2014 International Conference on Generative Programming: Concepts and Experiences* (*GPCE 2014*), pp.123–132, ACM (2014).

[13] Jensen, S.H., Møller, A. and Thiemann, P.: Type Analysis for JavaScript, *Proc. Static Analysis, 16th International Symposium* (*SAS 2009*), Lecture Notes in Computer Science, Vol.5673, pp.238–255, Springer (2009).

[14] Latendresse, M.: Automatic Generation of Compact Programs and Virtual Machines for Scheme, *Proc. Workshop on Scheme and Functional Programming* (2000) (online), available from ⟨http://www.ai.sri.com/~latendre/pli.pdf⟩.

[15] Marr, S., Daloze, B. and Mössenböck, H.: Cross-Language Compiler Benchmarking—Are We Fast Yet?, *Proc. 12th Symposium on Dynamic Languages* (*DLS 2016*), pp.120–131, ACM (2016).

[16] Monat, R., Ouadjaout, A. and Miné, A.: Static Type Analysis by Abstract Interpretation of Python Programs, *34th European Conference on Object-Oriented Programming* (*ECOOP 2020*), LIPIcs, Vol.166, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp.17:1–17:29 (2020).

[17] Onozawa, H., Iwasaki, H. and Ugawa, T.: Customizing JavaScript Virtual Machines for Specific Applications and Execution Environments (in Japanese), *Computer Software*, Vol.38, No.3, pp.23–40 (2021).

[18] Onozawa, H., Ugawa, T. and Iwasaki, H.: Fusuma: Double-ended threaded compaction, *Proc. 2021 ACM SIGPLAN International Symposium on Memory Management* (*ISMM 2021*), pp.94–106, ACM (2021).

[19] Shivers, O.: Control-Flow Analysis in Scheme, *Proc. ACM SIGPLAN'88 Conference on Programming Language Design and Implementation* (*PLDI 1988*), pp.164–174, ACM (1988).

[20] Tobin-Hochstadt, S. and Felleisen, M.: The design and implementation of typed scheme, *Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL 2008*), pp.395–406, ACM (2008).

[21] Ugawa, T., Iwasaki, H. and Kataoka, T.: eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems, *J. Comput. Lang.*, Vol.51, pp.261–279 (2019).

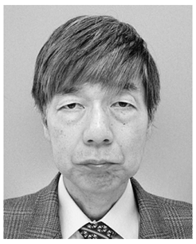
[22] Wimmer, C., Haupt, M., de Vanter, M.L.V., Jordan, M.J., Daynès, L. and Simon, D.: Maxine: An approachable virtual machine for, and in, java, *ACM Trans. Archit. Code Optim.*, Vol.9, No.4, pp.30:1–30:24 (2013).

[23] Wirfs-Brock, A. and Eich, B.: JavaScript: The first 20 years, *Proc. ACM Program. Lang.*, Vol.4, No.HOPL, pp.77:1–77:189 (2020).

[24] Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D. and Grimmer, M.: Practical partial evaluation for high-performance dynamic language runtimes, *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2017*), pp.662–676, ACM (2017).

[25] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D. and Wolczko, M.: One VM to rule them all, *Proc. ACM Symposium on New Ideas in Programming and Reflections on Software* (*Onward! 2013*), pp.187–204, ACM (2013).

Yuta Hirasawa received his M.E. degree from the University of Electro-Communications in 2022 and has been engaged in Fujitsu Limited since 2022. His research interests are programming languages.



Hideya Iwasaki is a professor in the School of Science and Technology at Meiji University, Japan. Until the end of March 2022, he had been a professor in the Graduate School of Informatics and Engineering at the University of Electro-Communications. He has been a member of the Science Council of Japan since 2011. He received an M.E. degree in 1985 and Dr.Eng. degree in 1988 from the University of Tokyo. His research interests includes programming languages and systems, parallel programming, systems software, and constructive algorithmics. He is a member of the IPSJ and ACM.

Tomoharu Ugawa received his B.Eng. degree in 2000, M.Inf. degree in 2002, and Dr.Inf. degree in 2005, all from Kyoto University. He worked for a research project on real-time Java at Kyoto University from 2005 to 2008. In 2008–2014, he was an assistant professor at the University of Electro-Communications in 2008–2014 and an associate professor at Kochi University of Technology in 2014–2020. He is currently an associate professor at the University of Tokyo. His work is in the area of implementation of programming languages with a specific focus on memory management. He received the IPSJ Yamashita SIG Research Award in 2012.

Hiro Onozawa received his M.E. degree from the University of Electro-Communications in 2022 and has been engaged in KLab Inc. since 2022. His research interests are JavaScript runtime systems and memory management.