eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems

Tomoharu Ugawa[a,*], Hideya Iwasaki[b], Takafumi Kataoka[a]

[a]*Kochi University of Technology, Tosayamada, Kami, Kochi 782-8502, Japan*
[b]*The University of Electro-Communications, Chofu, Tokyo 182-8585, Japan*

Abstract

Using JavaScript as a description language can increase the productivity of application programs on embedded systems. Since JavaScript is a dynamic language, it is important for a JavaScript virtual machine (VM) to efficiently identify the types of first-class values and perform type-based dispatches. For embedded systems, the type-based dispatching code is desired to be compact as well as fast. Although operators of JavaScript are heavily overloaded and capable of accepting a variety of datatypes as operands, all the datatypes are not always necessarily used in a specific program. If some datatypes are never used in this program, a VM for a subset of JavaScript with operators that support limited datatypes suffices. Such a VM is expected to be compact and efficient. In addition, internal representation of datatype of each value may affect performance of type-based dispatching. This paper presents a novel framework that can generate a VM for a subset of JavaScript on the basis of operand specifications and datatype specifications given by the programmer. The operand specifications describe possible operand datatypes for every instruction and the datatype specifications describe adequate internal representations of necessary datatypes for a target program. The generated VM is specialized in the sense that it has efficient and minimum type-based dispatching code for all instructions.

Keywords: virtual machine, JavaScript, embedded systems, code generation, decision diagram

[☆]This paper is a revised version of the paper entitled "A Framework for Constructing JavaScript Virtual Machines with Customized Datatype Representations" presented at SAC 2018.

[*]Corresponding author

Email addresses: `ugawa.tomoharu@kochi-tech.ac.jp` (Tomoharu Ugawa), `iwasaki@cs.uec.ac.jp` (Hideya Iwasaki), `kataoka@pl.info.kochi-tech.ac.jp` (Takafumi Kataoka)

1. Introduction

Embedded systems [1] are now becoming quite common. We can find various kinds of small devices installed with application programs that carry out certain specialized operations of their own and transmit, if necessary, data to other devices. Such application programs are typically developed in low-level languages such as C and assembly languages. However, program development by using these low-level languages suffers from three main productivity problems.

- Since low-level languages are not suitable for rapid prototyping, it takes much time to evolve applications on embedded systems.
- Writing programs in such languages is very cumbersome. For example, programmers have to be very careful in not forgetting to free allocated memory when it is no longer needed to avoid memory leak.
- Programs written in low-level languages have a problem in portability. Even though a program is written in C, it is less portable compared to a program written in a high-level language, e.g., JavaScript.

This paper focuses on the use of *virtual machine (VM) based managed languages* to resolve these problems. Out of a number of managed languages, we adopted JavaScript, which is one of widely-used scripting languages with a higher level of abstractions than low-level languages, for the following two main reasons.

- JavaScript is quite suitable for the rapid development of prototype programs from the inherent characteristics of scripting languages. This enables programmers to readily try and improve new ideas.
- JavaScript's event-driven programming style matches data processing in embedded systems [2, 3]. For example, an embedded system with a small device forms a sensor node in a network and collects data that are to be transmitted to another node in an event-driven manner. This kind of event-driven behavior can be naturally described as a JavaScript program.

The goal of this research is to make a JavaScript VM available on embedded systems. Such a VM is required to be compact and lightweight. This is because the capabilities of devices for embedded systems are generally "poor" in the sense that minimum required performance suffices for the objectives of the system. Over performance of the hardware of a device, e.g., excessive CPU speed or too much capacity in the installed memory, is avoided to reduce the cost of embedded systems, and especially in battery-powered systems, to reduce the power consumption.

Our approach is to allow programmers of embedded system applications to customize their own VMs for their individual applications. We have decided to take this approach on the basis of the following observations.

First, once an application is installed on an embedded device and begins its operation, the embedded system becomes specialized for the operation because another application is not supposed to be run at the same time on the same device. In this point, JavaScript VMs for embedded systems have a big advantage over those for web browsers. Since VMs for web browsers must be capable of executing any programs efficiently, they adopt dynamic adaptation techniques such as just-in-time compilation [4]. In contrast, every VM for an embedded system can be specialized to the target application.

Second, a JavaScript VM in many cases is not necessarily the "full-set" one. The operators of JavaScript are heavily overloaded and capable of accepting a variety of datatypes as operands. For example, according to the specification of JavaScript [5], the plus (+) operator behaves as an addition operator for number operands, a concatenation operator for string operands, and either of the two for other operand datatypes depending on the results of type conversions of the operands. However, many applications use the plus operator with limited combinations of operand datatypes. For example, some application may not apply the plus operator to string operands at all. Needless to say, few applications apply the plus operator to a combination of boolean and array operands.

Third, datatypes mainly and frequently handled in an application are fixed depending on its specialized operation. For example, integer data might be mainly used in some systems, while arrays might be heavily used in other systems.

This paper provides a framework that offers a mechanism of generating *customized JavaScript VMs* to the programmers of embedded systems. "Customizability" within this context has two aspects: *datatype selection* and *datatype representation*.

Our framework can generate a VM for the aspect of datatype selection that excludes the interpreter code for handling datatypes that will never be given as an operand of each instruction. The feature of datatype selection contributes to reducing the size of VM code. Limiting possible datatypes also makes type-based dispatching process needed for operator overloading simple.

Our framework can also generate a VM for the aspect of datatype representation that has efficient internal representations for frequently used datatypes. For example, if the programmer specifies the use of a special format of tagged pointers for a program with a heavy use of arrays to distinguish arrays from other datatypes, the program can assess quickly whether a given value is an array or not.

The above idea was incorporated into our new JavaScript VM named eJSVM (embedded JavaScript Virtual Machine), and the framework named eJSTK (embedded JavaScript Tool Kit) offers a mechanism of customizability to the programmer for eJSVM with respect to the two previously described aspects. The programmer gives two descriptions: the *operand specifications* for datatype selection and the *datatype specifications* for customizing datatype representation. Given these descriptions, eJSTK produces a set of C source programs of the customized eJSVM. Then, by using a standard C compiler such as GCC and Clang, the programmer can obtain the executable file of the advantageously customized

VM. The customized VM is optimized for a specific application whose requirements are written in the given descriptions at the cost of giving up supporting other applications. If the application performs an operation that is beyond the given datatype specifications, the VM can crash.

Our framework currently supports a subset of ECMAScript 5.1 [5], from which some features that need complicated treatment by eJSVM, such as the `eval()` function, are excluded.

The main contributions of the work reported in this paper are summarized below.

- We present eJSTK, which is a framework that enables the programmer to generate a customized JavaScript VM. eJSTK is datatype-centric; the programmer gives the operand and datatype specifications, from which an efficient VM for specialized computations is generated.
- We present an algorithm that generates C code with a compact and efficient type-based dispatching for an interpreter from the operand and datatype specifications and the descriptions of VM instruction behaviors of the full-set JavaScript written in a domain specific language (DSL).
- We provide some experimental customizations of JavaScript VMs built from different operand and datatype specifications, and show the effectiveness of eJSTK.

2. eJSVM

2.1. Features of JavaScript

JavaScript is a multi-paradigm scripting language that has prototype-based object-oriented features [6]. It is widely accepted as a description language for client-side programs of Web applications. JavaScript is currently also used in server-side website programming. Well-known JavaScript engines include V8[1], Rhino[2], SpiderMonkey[3], V7[4], and JerryScript[5].

Table 1 presents datatypes in JavaScript. Types determined by the language specification [5] are listed in the left column of the table. Of these six types, only an Object has a collection of properties. The Object type can be further classified into simple Object, Array, Function, Regexp, etc., as indicated in the middle column. Thus, in essence, we can consider that JavaScript's types are those listed in the third column of the table. We call them *JS-datatypes* in the rest of this paper. We also use the term *VM-datatypes* to represent internal datatypes for implementing JS-datatypes in a VM. VM-datatypes defined in the eJSVM will be explained in Sect. 2.2.

[1]https://developers.google.com/v8/
[2]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[3]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals
[4]https://github.com/cesanta/v7
[5]http://jerryscript.net/

Table 1: Types of JavaScript.

types in specification	classification of Object	JS-datatype	description
Undefined		Undefined	`undefined`
Null		Null	`null`
Boolean		Boolean	`true` or `false`
String		String	string
Number		Number	number
Object	Object Object	SimpleObject	normal object
	Array	Array	array
	Function	Function	function
	Regexp	Regexp	regular expr
	Boolean Object	BooleanObject	boxed boolean
	String Object	StringObject	boxed string
	Number Object	NumberObject	boxed number

Since JavaScript is a dynamic language, executing a JavaScript program causes many type-based dispatches according to the values of interest and also many type conversions from a value of some type to another value of another type. This is partly because some operators work differently depending on the types of their operands, and partly because some operators convert the types of their operands to those they expect.

For example, consider the addition operator "+". It behaves as follows.

- When the values of both operands are numbers, it returns their sum as a number.
- When the values of both operands are strings, it returns the concatenated string.
- When either of the values of operands is a string or an object that can be converted to a string, it converts the other operand to a string and returns the concatenated string.
- Otherwise, it converts both operands to numbers and returns their sum.

As can be seen from this example, it is important for a JavaScript VM to efficiently identify the datatype of a given value and dispatch its execution on the basis of the identified datatype.

2.2. Internals of eJSVM

The eJSVM is a register-based virtual machine for JavaScript programs that are aimed to be run on embedded systems. It is an interpreter of VM instructions, into which JavaScript programs are compiled by the *eJS compiler.* The main loop of eJSVM executes a compiled program by interpreting a sequence of VM instructions one by one on the basis of the threaded code [7] technique.

The instruction set of eJSVM was designed by the authors. We explain the `add` instruction that corresponds to the "+" operator as a typical example of

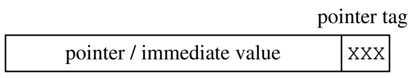
Table 2: JS-datatypes, VM-datatypes, and VM-reptypes in default setting of eJSVM.

JS-datatype	VM-datatype	VM-reptype	ptag/htag	ptr/imm	description
Undefined	special	normal_special	110 / –	imm	
Null	special	normal_special	110 / –	imm	
Boolean	special	normal_special	110 / –	imm	
String	string	normal_string	100 / 4	ptr	seq of chars
Number	fixnum	normal_fixnum	111 / –	imm	61-bit signed int
	flonum	normal_flonum	101 / 5	ptr	C's double
SimpleObject	simple_object	normal_simple_object	000 / 6	ptr	normal object
Array	array	normal_array	000 / 7	ptr	array
Function	function	normal_function	000 / 8	ptr	user-defined
	builtin	normal_builtin	000 / 9	ptr	built-in
Regexp	regexp	normal_regexp	000 / 11	ptr	regular expr
BooleanObject	boolean_object	normal_boolean_object	000 / 14	ptr	boxed boolean
StringObject	string_object	normal_string_object	000 / 12	ptr	boxed string
NumberObject	number_object	normal_number_object	000 / 13	ptr	boxed number

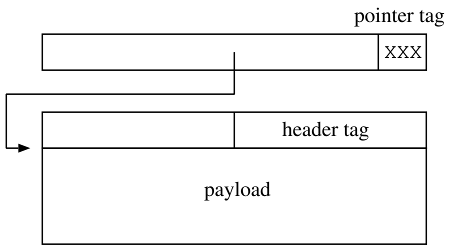
instruction execution. This instruction takes three registers as its operands; the first is the destination register of the result, and the second and third are input registers that have the values of the augend and addend respectively. Interpreting this instruction, the VM checks the datatypes of these input values and appropriately dispatches the execution according to the combination of the datatypes to perform adequate type-based operations of "+". Since JavaScript is a dynamic language, many VM instructions need similar dispatching processes at runtime. In fact, 26 out of 59 VM instructions need such dispatching.

The proposed framework defines internal datatypes called *VM-datatypes* in the eJSVM that correspond to JS-datatypes. Table 2 lists VM-datatypes. Basically, the relationship between JS-datatypes and VM-datatypes is one-to-one, but there are three exceptions. First, Undefined, Null, and Boolean correspond to special. Since these JS-datatypes have very small number of instances, i.e., Undefined and Null have only one (`undefined` and `null`, respectively) and Boolean has only two (`true` and `false`), we have decided to treat them altogether in a single VM-datatype special. Second, Number has two VM-datatypes, namely fixnum (signed integers) and flonum (floating point numbers). It is natural to have these two, one of which is specialized to integers, because integer values are very frequently used. The third exception is Function. Since an instance of a Function is either a user-defined or a built-in one, eJSTK defines two VM-datatypes. The relationship between JS-datatypes and VM-datatypes is fixed; the programmer is not allowed to change the relationship in customization.

Each customization has to provide concrete representations for necessary VM-datatypes. We call a representation of a VM-datatype a *VM-datatype representation*, or *VM-reptype* for short. Defining VM-reptypes is very important in the customization of eJSTK. The default setting of eJSVM defines VM-reptypes (third column of Table 2) so that every VM-datatype has a single VM-reptype. VM-reptypes in the default setting are predefined. Thus, programmers can make use of these VM-datatypes in their own customization.



(a) Tagged pointer.



(b) Header tag in data structure.

Figure 1: Tagged pointer and header tag in eJSVM.

The eJSVM internally uses *pointer tags* in *tagged pointers* and *header tags* simultaneously to identify the VM-reptype (and thus VM-datatype) of a first-class value in JavaScript, on the condition that every data structure is n-byte aligned in the heap area where n is determined by the architecture of the CPU. For the case of a 64-bit architecture, n is eight and thus the least significant three bits of an aligned address are definitely zero. These bits can be used to hold a pointer tag value to represent its VM-reptype. At most, eight distinct pointer tag values can be used in this case. For the case where eight tag values are insufficient to identify VM-reptypes, eJSVM also uses a separate tag field, i.e., header tag, in the body of a data structure in the heap. We assume that the pointer tag has three bits in the 64-bit architecture in the rest of this paper because eJSVM currently supports only such architectures. However, the design of eJSTK is independent of the width of a pointer tag.

Figure 1 outlines a tagged pointer and a header tag in the eJSVM. A first-class value is represented as a tagged pointer that contains either a pointer to a data structure or an immediate value when a pointer is unnecessary, such as an integer value. If it has a pointer, the pointed data structure has a header within which the header tag is placed. The right of Table 2 presents pointer tags and header tags for VM-reptypes in the default setting of the eJSVM.

If a VM-reptype uses a pointer tag value that is not shared with another VM-reptype, e.g., normal_string in Table 2, whether the value of a given tagged pointer is of the VM-reptype or not can be quickly determined by only checking its pointer tag. We say that such a pointer tag is *unique*. In contrast, if a VM-reptype's pointer tag value is not unique, i.e., *shared* with another VM-reptype (in the case the pointer tag has a pointer in it), it requires two checks for both the pointer tag value and the header tag value to determine whether a value is of the VM-reptype. This is costly compared to the first case because it needs indirect access to the header tag. Thus, it is important to assign a unique pointer tag value for a frequently used VM-reptype to gain the efficiency of the type-based dispatch in a customized VM. It is worth noting that every VM-

```
#define T_STRING  4           // pointer tag for string
#define T_GENERIC 0           // pointer tag for simple_object/array
#define HTAG_SIMPLE_OBJECT 6 // header tag for simple_object
#define HTAG_ARRAY 7          // header tag for array
// individual VM-datatype
#define is_string(v) (PTAG(v) == T_STRING)
#define is_simple_object(v) \
    ((PTAG(v) == T_GENERIC) && (HTAG(v) == HTAG_SIMPLE_OBJECT))
#define is_array(v) ((PTAG(v) == T_GENERIC) && (HTAG(v) == HTAG_ARRAY))
// set of VM-datatypes
#define is_object(v) (PTAG(v) == T_GENERIC)
#define is_number(v) ((PTAG(v) == T_FIXNUM) || (PTAG(v) == T_FLONUM))
```

Figure 2: Predicate macros for default setting.

reptype that has an immediate value in its tagged pointer is necessarily assigned a unique pointer tag. normal_fixnum is an example of such VM-reptypes in the default setting shown in Table 2. One might think that the header tag for a VM-reptype that is assigned a unique pointer tag value is unnecessary because it is never referred to in identifying the datatype. However, it is retained in preparation for cases where garbage collection[6] requires the header tag value of every data structure in the heap area.

The C source code of eJSVM needs *predicate macros*. A predicate macro has the form of `is_`*vmdatatype*`(v)`, where *vmdatatype* is the name of a VM-datatype, and returns whether a tagged pointer `v` is *vmdatatype* or not. Predicate macros of this type are called *VM-datatype predicates*. In addition, eJSVM needs predicate macros for a set of VM-datatypes, which are called *datatype family predicates*. For example, `is_object(v)` returns whether `v` is one of the VM-datatypes corresponding to Object (see Table 1). Figure 2 presents some predicate macros for the default setting listed in Table 2. Predicate macros are automatically defined by eJSTK for a customized VM. Here, `PTAG(v)` returns the pointer tag value and `HTAG(v)` returns header tag value of a given tagged pointer `v`.

3. Overview of proposed framework

The proposed framework, eJSTK, offers the programmer means of making datatype-centric customizations, i.e., datatype selection and datatype representation. Due to these customizations, the size of program code of the customized eJSVM decreases because unnecessary cases concerning unused datatypes are not generated. This is crucial from the viewpoint of limited resources in embedded systems. Furthermore, the type-based dispatching process in the customized eJSVM becomes simplified and the VM is expected to be speeded up.

Figure 3 presents the overall structure of eJSTK.

[6]eJSVM currently implements mark and sweep garbage collection.

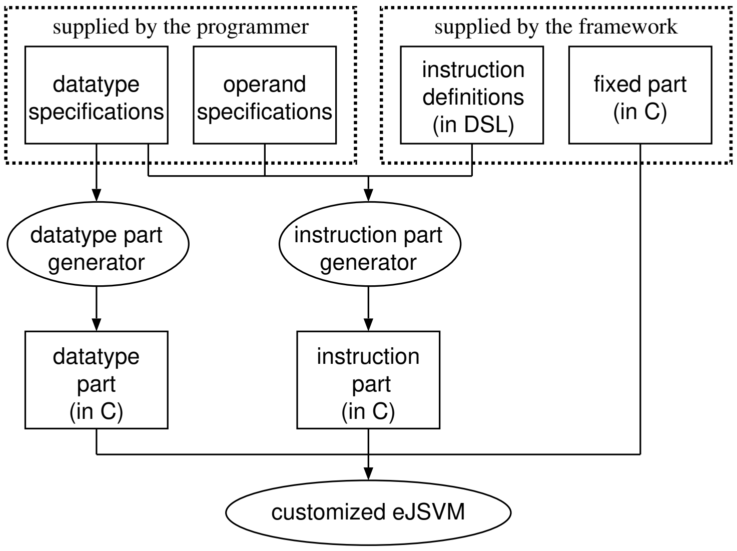


Figure 3: Overall structure of eJSTK.

The programmer has to properly specify the following *datatype specifications* to customize a VM:

1. correspondences between VM-datatypes and VM-reptypes, and
2. assignments of pointer tag values and header tag values to VM-reptypes.

In the design of eJSTK, the relationships between VM-datatypes and VM-reptypes are not limited to one-to-one; the programmer is allowed to define more than one VM-reptype for a VM-datatype as long as every VM-reptype is implemented in the fixed part in Fig. 3. The reason why we took this design decision is that we have a plan to enable the programmer to give multiple VM-reptypes to string as JerryScript does. In JerryScript, short strings are represented as *direct strings*, which have a totally different structure from that of normal strings; a normal string is allocated in the heap while a direct string is embedded within a pointer. Although the current eJSTK has not implemented direct strings in the fixed part yet, the datatype part generator and the instruction part generator can generate type-based dispatching code for those VM-datatypes that have multiple VM-reptypes. We will give an example in Sect. 4.3.

In addition to datatype specifications, the programmer has to specify *operand specifications* for every instruction that describe possible datatypes given as its operand. Instructions that are not given operand specifications are supposed to take all datatypes as default.

The entire C source code of eJSVM can be grouped into three categories. First, the *fixed part* consists of code that is independent of customization. This is supplied by the framework and the programmer does not need to know its details. Second, the *datatype part* consists of code generated by the *datatype*

```
special:                +normal_special
string:                 +normal_string
fixnum:                 +normal_fixnum
flonum:                 +normal_flonum
simple_object:          +normal_simple_object
array:                  +normal_array
function:               +normal_function
builtin:                +normal_builtin
regexp:                 +normal_regexp
string_object:          +normal_string_object
number_object:          +normal_number_object
boolean_object:         +normal_boolean_object

normal_special:         T_SPECIAL(110)
normal_string:          T_STRING(100)/HTAG_STRING(4)
normal_fixnum:          T_FIXNUM(111)
normal_flonum:          T_FLONUM(101)/HTAG_FLONUM(5)
normal_simple_object:   T_GENERIC(000)/HTAG_SIMPLE_OBJECT(6)
normal_array:           T_GENERIC(000)/HTAG_ARRAY(7)
normal_function:        T_GENERIC(000)/HTAG_FUNCTION(8)
normal_builtin:         T_GENERIC(000)/HTAG_BUILTIN(9)
normal_regexp:          T_GENERIC(000)/HTAG_REGEXP(11)
normal_string_object:   T_GENERIC(000)/HTAG_STRING_OBJECT(12)
normal_number_object:   T_GENERIC(000)/HTAG_NUMBER_OBJECT(13)
normal_boolean_object:  T_GENERIC(000)/HTAG_BOOLEAN_OBJECT(14)
```

Figure 4: Datatype specifications for default setting.

part generator in eJSTK on the basis of given datatype specifications. Its main content consists of definitions of predicate macros. Third, the *instruction part* consists of code for executing VM instructions in the interpreter's main loop. This part is generated by the *instruction code generator* in eJSTK from the instruction definitions, datatype specifications and operand specifications. Here, the *instruction definition* is supplied by the framework; it specifies the type-based behavior of every VM instruction.

The eJSVM uses VM-datatypes in four places: in the interpreter's main loop that executes a sequence of VM instructions, in datatype conversion functions, in built-in functions, and in the tracer of the garbage collector. The current eJSTK generates efficient dispatcher only for the interpreter of VM instructions. Generating type-based dispatching code for the other places is left for our future work.

4. Specifying customizations

4.1. Datatype specifications

We first present datatype specifications for the default setting of eJSVM in Fig. 4. Each line consists of two parts separated by a colon. The left part of the colon is a name of either a VM-datatype or a VM-reptype. The names of VM-datatypes (e.g., string and array) and those of VM-reptypes for the default setting (e.g., normal_string and normal_array) are reserved.

When a VM-datatype emanates in the left part, its corresponding VM-reptypes occur in the right part with a "+" character in front of each VM-reptype. Lines 1–12 in the example of Fig. 4 mean that every VM-datatype is only represented by its default VM-reptype.

When a VM-reptype emanates on the left, assignments of pointer tag values and header tag values occur on the right in the following form.

ptag_name`(`*ptag_value*`)/`*htag_name*`(`*htag_value*`)`

where *ptag_name* is the name for the pointer tag value, *ptag_value* is the pointer tag value in a binary number, *htag_name* is the name for the header tag value, and *htag_value* is the header tag value in a decimal number. Both *ptag_name* and *htag_name* are used by the macro definitions in the generated datatype part. The "/" or later is only necessary when the tagged pointer has a pointer within it[7]. For example, Line 16 in Fig. 4 indicates that the VM-reptype normal_fixnum has 111 as its pointer tag with immediate data in its tagged pointer. Similarly, Line 19 means that the VM-reptype normal_array has 000 as its pointer tag and its tagged pointer points to a data structure whose header tag is seven.

eJSTK generates macro definitions, part of which have already been presented in Fig. 2, from the datatype specifications.

4.2. Operand specifications

Though the language specification [5] determines type conversion rules for each JavaScript's operator, it is likely that an application only gives values of some limited datatypes for each VM instruction. For example, an application only gives numbers as operands to every occurrence of the `add` instruction. To customize the eJSVM in such a case, the programmer gives *operand specifications* that describe possible datatypes given to each VM instruction.

Each line in operand specifications has the following form.

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

Here, "`(`*operand_type*`, ...)`" specifies a combination of datatypes for input operands, where *operand_type* is either a name of VM-datatype or VM-reptype, "_" (which means any datatype), or "`-`" (which means that the operand is not an input), and *action* is either `accept`, `error`, or `unspecified`.

In the default setting, every instruction is allowed to be given all datatypes as its input operand. Thus, its operand specifications, part of which are presented in Fig. 5, do not limit input datatypes at all.

4.3. Examples of customization

We here explain how to specify a required customization by providing two examples.

Example 1: Assigning a unique pointer tag to Array

Suppose that we want to run an application that uses many arrays. This example customizes the eJSVM to have a special treatment for the Array type. The VM-reptype normal_array has a pointer tag value of 000 in the default

[7] In this case, the structure name for the body pointed to from a tagged pointer has to be specified on the same line. However, we omit it in this paper for simplicity.

```
add (-,_,_) accept       // instruction for '+'
bitand (-,_,_) accept    // instruction for '&'
bitor (-,_,_) accept     // instruction for '|'
call (_,-) accept        // instruction for function call
div (-,_,_) accept       // instruction for '/'
eq (-,_,_) accept        // instruction for '=='
// The rest is omitted
```

Figure 5: Operand specifications for default setting.

```
18      (Lines 1–18 are the same as those of Fig. 4)
19 normal_array:          T_ARRAY(010)/HTAG_ARRAY(7)
20      (Lines 20–25 are the same as those of Fig. 4)
```

Figure 6: Datatype specifications for Example 1.

setting. This pointer tag value is shared with JavaScript's Object family, such as simple_object and function. This example assigns a unique pointer tag value of 010 to normal_array and allows normal_arrays to be quickly examined.

Figure 6 presents the datatype specifications for this customization. The only difference from the specifications of default setting is on Line 19. normal_array in this example is assigned a unique pointer tag of 010. The operand specifications for this customization are the same as those for default setting.

Example 2: Limiting operand datatypes

Suppose that we are developing an application that uses only limited combinations of operand datatypes for some instructions, especially which perform arithmetic / comparison operations. For example, the add instruction in this application assumes that both operands are fixnums, both are flonums, augend is flonum and addend is fixnum, or both are strings.

Figure 7 presents part of the operand specifications for this application. This says that the above combinations for add are accepted but other combinations should signal errors. Any datatype specifications can be used with the operand specifications in Fig. 7.

Example 3: Making direct string

Our last example is to make two kinds of VM-reptypes for string VM-datatype. In the default setting, VM-reptype for string is normal_string. Its tagged pointer has an address of a data structure allocated in the heap area, which contains a null-terminated C string in it.

Suppose an application that uses many short strings. If a tagged pointer has

```
add (-,fixnum,fixnum) accept
add (-,flonum,fixnum) accept
add (-,flonum,flonum) accept
add (-,string,string) accept
add (-,_,_) error
```

Figure 7: Part of operand specifications for Example 2.

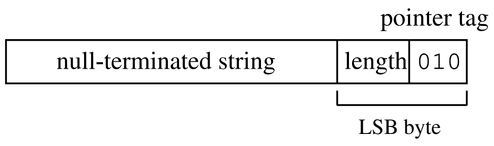


Figure 8: Tagged pointer for direct_string.

```
 1 special:                +normal_special
 2 string:                 +normal_string +direct_string
 3 fixnum:                 +normal_fixnum
 4 flonum:                 +normal_flonum
 5 simple_object:          +normal_simple_object
 6 array:                  +normal_array
 7 function:               +normal_funcition
 8 builtin:                +normal_builtin
 9 regexp:                 +normal_regexp
10 string_object:          +normal_string_object
11 number_object:          +normal_number_object
12 boolean_object:         +normal_boolean_object
13
14 normal_special:         T_SPECIAL(110)
15 normal_string:          T_STRING(100)/HTAG_STRING(4)
16 direct_string:          T_DSTRING(010)
17 normal_fixnum:          T_FIXNUM(111)
18 normal_flonum:          T_FLONUM(101)/HTAG_FLONUM(5)
19 normal_simple_object:   T_GENERIC(000)/HTAG_SIMPLE_OBJECT(6)
20 normal_array:           T_GENERIC(000)/HTAG_ARRAY(7)
21 normal_function:        T_GENERIC(000)/HTAG_FUNCTION(8)
22 normal_builtin:         T_GENERIC(000)/HTAG_BUILTIN(9)
23 normal_regexp:          T_GENERIC(000)/HTAG_REGEXP(11)
24 normal_string_object:   T_GENERIC(000)/HTAG_STRING_OBJECT(12)
25 normal_number_object:   T_GENERIC(000)/HTAG_NUMBER_OBJECT(13)
26 normal_boolean_object:  T_GENERIC(000)/HTAG_BOOLEAN_OBJECT(14)
```

Figure 9: Datatype specifications of Example 3.

64 bits, a null-terminated short string whose length is less than seven can be packed into a tagged pointer as an immediate data. Although this representation for short string is not implemented yet, in the current eJSTK, this example assumes that it were implemented.

This customization example makes two VM-reptypes, namely direct_string[8] (for short strings) and normal_string (for other strings), as representations of string VM-datatype. We use 010 as the pointer tag value of direct_string.

Figure 8 presents the tagged pointer for direct_string. Before the last byte comes a sequence of characters. The last byte contains the pointer tag (least significant three bits) and the length of the string.

The datatype specifications are presented in Fig. 9. Line 2 specifies that VM-datatype string has two VM-reptypes, normal_string and direct_string. Line 16 assigns a pointer tag 010 to direct_string. Since tagged pointer for direct_string contains immediate data, i.e., null-terminated string and length information,

[8] This is a different representation from the direct string of JerryScript.

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

Figure 10: Syntax of DSL for instruction definition.

header tag is not, of course, assigned.

5. Instruction definitions

The eJSTK generates C code for VM instruction execution in the main loop of the eJSVM interpreter by using instruction definitions, datatype specifications and operand specifications.

eJSTK supplies an instruction definition written in DSL for every instruction. As presented in Fig. 3, the instruction definition is referred to only by the instruction part generator. Thus, programmers need not be familiar with the DSL, and furthermore need not write a "program" in this DSL.

The syntax for the DSL is presented in Fig. 10.

Figure 11 presents the instruction definition written in DSL for the `add` instruction. Its three operands are given as register numbers, but its instruction definition supposes that the place of the first operand (destination register) has already been stored into `dst` of type `Register`, and the values retrieved from the second and third registers have already been stored into `v1` (augend) and `v2` (addend) of type `Value`. This definition describes eight cases for type-based dispatching.

The DSL describes the behavior of the instruction as a collection of when-clauses, each of which is composed of a condition on the VM-datatypes of its input operands and a C code fragment that is to be executed when the condition is satisfied. For example, a when-clause is described in Lines 3–9 of Fig. 11; its condition states that both `v1` and `v2` are fixnums, and if this condition is satisfied, a C code fragment at Lines 4–8 is executed. Types specified in a condition part of a when-clause are not VM-reptypes but VM-datatypes because every instruction definition is independent of customization, i.e., concrete

```
\inst add (Register dst, Value v1, Value v2)
      // v1 and v2 have values in the source registers.
\when v1:fixnum && v2:fixnum \{
  cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
           // Adds two Fixnums after converting them into C integer.
  dst = cint_to_number(s);
        // Stores the sum to the destination register after converting the sum
        // to a JavaScript's number.
\}
\when v1:string && (v2:fixnum || v2:flonum || v2:special) \{
  v2 = to_string(context, v2);          // First converts v2 to a string.
  goto add_STRSTR;                      // Then goes to add_STRSTR.
\}
\when (v1:fixnum || v1:flonum || v1:special) && v2:string \{
  v1 = to_string(context, v1);          // First converts v1 to a string.
  goto add_STRSTR;                      // Then goes to add_STRSTR.
\}
\when v1:string && v2:string \{
add_STRSTR:            // This label is the case where both values are strings.
  dst = cstr_to_string2(context, string_to_cstr(v1), string_to_cstr(v2));
        // Concatenating two strings after converting them into C strings.
\}
\when (v1:simple_object || v1:array || v1:function || v1:builtin || v1:regexp
       || v1:string_object || v1:number_object || v1:boolean_object) &&
      (v2:special || v2:string || v2:fixnum || v2:flonum) \{
  v1 = object_to_string(context, v1);   // First converts v1 to a string.
  goto add_HEAD;                        // Then goes to the entrance.
\}
\when (v1:special || v1:string || v1:fixnum || v1:flonum) &&
      (v2:simple_object || v2:array || v2:function || v2:builtin || v2:regexp
       || v2:string_object || v2:number_object || v2:boolean_object) \{
  v2 = object_to_string(context, v2);   // First converts v2 to a string.
  goto add_HEAD;                        // Then goes to the entrance.
\}
\when (v1:simple_object || v1:array || v1:function || v1:builtin || v1:regexp
       || v1:string_object || v1:number_object || v1:boolean_object) &&
      (v2:simple_object || v2:array || v2:function || v2:builtin || v1:regexp
       || v2:string_object || v2:number_object || v2:boolean_object) \{
  v1 = object_to_string(context, v1);   // Converts v1 to a string.
  v2 = object_to_string(context, v2);   // Converts v2 to a string.
  goto add_HEAD;                        // Then goes to the entrance.
\}
\otherwise \{
  double x1 = to_double(context, v1);   // Converts v1 to a C double.
  double x2 = to_double(context, v2);   // Converts v2 to a C double.
  dst = double_to_number(x1 + x2);
        // Adds them, converts the sum to a number, and stores it to the destination.
\}
```

Figure 11: Instruction definition for `ADD` instruction.

representations of VM-datatypes. If none of the conditions in when-clauses are satisfied, C code fragment in the otherwise-clause, if any, is executed.

It is worth noting that the order of when-clauses in an instruction definition is not in the least important. The instruction part generator adequately arranges the order of condition checking and C code fragments for every pair of interest by using the information in the datatype specifications to generate the instruction part. How to generate C code from an instruction definition will be described in Sect. 7.

Second, the DSL enables the C code fragment of an instruction to easily access all given operands via its corresponding formal parameters. Each formal parameter has a "type" which is determined depending on its role; `Register` for a place of a register, `Value` for a value (tagged pointer) stored in an input register, `Subscript` for an integer representing a subscript, `Immediate` for an immediate value of a fixnum/special, and `Displacement` for a displacement for a jump/conditional jump instruction.

For the sake of convenience, the DSL offers the prologue and epilogue parts, which are inserted at the entrance and exit of the generated instruction code, respectively. Though they are not used in Fig. 11, their typical usages might be to define comprehensive names by `#define` in the "prologue" part and undefine them by `#undef` in the "epilogue" part.

The proposed DSL does not allow the nesting of conditionals and C code fragments. As long as our definitions of VM instructions, nesting was not necessary; we used `goto` statements to perform further type-based dispatching after converting some operands as shown in Fig. 11. The `goto` statements are used not only for representing nesting but also for unifying similar type-based dispatching code.

6. Generating customized code for datatype part

From the datatype specifications, eJSTK generates macro definitions, some of which have already been presented in Fig. 2. eJSVM uses two kinds of predicate macros.

- A *VM-datatype predicate macro* returns whether a given value is of a VM-datatype of interest.
- A *datatype family predicate macro* returns whether a given value is of a VM-datatype that belongs to a particular group of VM-datatypes.

A VM-datatype may correspond to multiple VM-reptypes. For example, in Example 3 in Sect. 4.3, the string VM-datatype corresponds to two VM-reptypes, normal_string and direct_string. Thus, not only a datatype family predicate macro but also a VM-datatype predicate macro judges whether the VM-reptype of a given value belongs to a certain set of VM-reptypes.

The program of the runtime system is hand-written "static" code and is independent of datatype representations. This makes it possible for the programmer to customize datatype representations. Whenever the runtime system examines the VM-datatype of a value, it uses a VM-datatype predicate macro or a datatype family predicate macro. For example, built-in functions often use a `is_object` datatype family predicate macro to judge whether a given value belongs to the JavaScript's Object family, such as simple_object and array.

Every VM-datatype predicate macro and datatype family predicate macro can be defined as a disjunction of predicates, each of which returns whether a given value is of a specified VM-reptype. However, if we simply defined them in this way, predicate macros would be inefficient. For example, `is_object` with a default datatype specification defined as

```
#define is_object(v) \
  (is_simple_object(v) || is_array(v) || ...)
```

is expanded to the following.

```
#define is_object(v) \
  ((PTAG(v) == T_GENERIC) && (HTAG(v) == HTAG_SIMPLE_OBJECT) \
   || (PTAG(v) == T_GENERIC) && (HTAG(v) == HTAG_ARRAY) \
   || ...)
```

Instead, eJSTK simplifies such a macro definition by using the following rules.

- Combine terms with the same test for a pointer tag value by using the distributive law.
- Replace a disjunction that covers all possible cases with `TRUE`.

In the case of `is_object`, all terms have `(PTAG(v) == T_GENERIC)` in common. Thus, eJSTK transforms the definition by combining all terms as follows.

```
#define is_object(v) \
  ((PTAG(v) == T_GENERIC) && \
   ((HTAG(v) == HTAG_SIMPLE_OBJECT) || \
    (HTAG(v) == HTAG_ARRAY) || ...))
```

Then, since the disjunction of tests for a header tag of `v` covers all possible header tag values, eJSTK transforms it into the following definition.

```
#define is_object(v) ((PTAG(v) == T_GENERIC) && TRUE)
```

The latter transformation is impossible for a C compiler because it cannot know if a disjunction covers all possible cases. We apply the former transformation to enhance the opportunity for the latter transformation. Please note that it is not necessary to eliminate `TRUE` because it will be eliminated by the C compiler.

7. Generating customized code for instruction part

An instruction needs to dispatch its execution code with respect to the datatypes of the given input values, which have already been stored in its operands. The instruction part generator of eJSTK outputs *rearranged* C code for this purpose, in the sense that the code excludes on the basis of operand specifications unnecessary cases for unused datatypes. Since the condition part in an instruction definition is written by using VM-datatypes, eJSTK re-interprets each VM-datatype to its corresponding VM-reptypes and generates C code that identifies these VM-reptypes.

We prioritize both the efficiency and compactness of the generated C code. For the sake of efficiency, the C code accesses the header tag of a data structure only when the pointer tag value in a tagged pointer is shared. We assume that VM-reptypes with a unique pointer tag value are used with almost the same frequency. We understand that this assumption is not always practical, and leave its improvement to our future work. We avoid duplicating the same C code fragment for compactness by using `goto` statements when necessary.

eJSTK uses a decision diagram as its internal representation of type-based dispatching code. It generates rearranged C code for an instruction using a four-step process.

1. Read the instruction definition written in the DSL (e.g., Fig. 11) and convert it into a collection of *normalized dispatch rules*, each of which specifies a VM-reptype for each operand and the C code fragment to be executed if datatypes of operands match the specified VM-reptypes.
2. Construct a decision tree from the instruction definition. A leaf of the tree corresponds to a C code fragment in the instruction definition. The internal nodes of the tree are constructed from the condition part. Each of the internal nodes corresponds to a multi-way branch based on the pointer tag value or the header tag value of an operand.
3. Optimize the decision tree. In this step, some nodes might be combined and shared with multiple parent nodes. As a result, the decision tree is transformed into a directed acyclic graph (DAG).
4. Generate C code from the DAG.

7.1. Running example

We explain the details of the code generation process by using a running example. Although the example is simple and rather artificial, it suffices to outline the process.

Suppose that we are interested in only three VM-datatypes, namely fixnum, string, and array. We use the default internal representations for both fixnum and array: normal_fixnum and normal_array. We use two VM-reptypes for string, normal_string and direct_string, which are introduced in Sect. 4.3. The datatype specifications for this example are presented in Fig. 12 (a). Different from Example 3 in Sect. 4.3, we assign the same pointer tag value to normal_array and normal_string. Note that both `T_DSTRING` and `T_FIXNUM` are unique and that `T_GENERIC` is shared.

We generate instruction code for a two-operand instruction, `exam_insn`, which is not an actual instruction. Its instruction definition is presented in Fig. 12 (b). We use `A`, `B`, `C`, `D`, and `E` instead of real C code fragments for simplicity. We assume that the values of operands are already stored in `v0` and `v1`.

```
string:              +normal_string +direct_string
fixnum:              +normal_fixnum
array:               +normal_array

normal_string:       T_GENERIC(000)/HTAG_STRING(4)
direct_string:       T_DSTRING(010)
normal_fixnum:       T_FIXNUM(111)
normal_array:        T_GENERIC(000)/HTAG_ARRAY(7)
```

(a) Datatype specifications.

```
\inst exam_insn (Value v0, Value v1)
\when v0:fixnum && v1:fixnum \{ A \}
\when v0:fixnum && (v1:string || v1:array) \{ B \}
\when (v0:string || v0:array) && v1:fixnum \{ C \}
\when v0:string && v1:string \{ D \}
\otherwise \{ E \}
```

(b) Instruction definition for example instruction.

Figure 12: Running example of code generation.

For the sake of brevity, we use the following abbreviated notations. DS, NS, NF, and NA denote VM-reptypes direct_string, normal_string, normal_fixnum, and normal_array, respectively. P_S, P_F, and P_G denote pointer tag values T_DSTRING, T_FIXNUM, and T_GENERIC, and H_S and H_A denote header tag values for normal_string and normal_array.

7.2. Code generation process

As mentioned above, eJSTK generates rearranged C code using a four-step process.

7.2.1. Step 1: Condition normalization

eJSTK transforms the instruction definition written in the DSL into a collection of normalized dispatch rules. A normalized dispatch rule is a pair of a condition and a C code fragment, where the condition part specifies a single VM-reptype for each operand. For example, if the instruction for our running example takes two operands, a condition is (t_0, t_1), which means that the VM-reptype of `v0` is t_0 and that of `v1` is t_1.

eJSTK decomposes a single ⟨when-clause⟩ or an ⟨otherwise-clause⟩ in an instruction definition into one or more normalized dispatch rules. At the same time, it discards some normalized dispatch rules that do not appear in the operand specifications. More specifically, eJSTK creates a set of normalized dispatch rules as follows.

1. eJSTK decomposes each condition into one or more *half-normalized* dispatch rules, where a half-normalized dispatch rule is a pair of a condition specifying a single VM-*data*type for each operand and a C code fragment. In this process, ⟨otherwise-clause⟩ is replaced with an appropriate collection of half-normalized dispatch rules.

(NF, NF)	$\Rightarrow$ `A`	(DS, NS)	$\Rightarrow$ `D`
(NF, DS)	$\Rightarrow$ `B`	(NS, DS)	$\Rightarrow$ `D`
(NF, NS)	$\Rightarrow$ `B`	(NS, NS)	$\Rightarrow$ `D`
(NF, NA)	$\Rightarrow$ `B`	(DS, NA)	$\Rightarrow$ `E`
(DS, NF)	$\Rightarrow$ `C`	(NS, NA)	$\Rightarrow$ `E`
(NS, NF)	$\Rightarrow$ `C`	(NA, DS)	$\Rightarrow$ `E`
(NA, NF)	$\Rightarrow$ `C`	(NA, NS)	$\Rightarrow$ `E`
(DS, DS)	$\Rightarrow$ `D`	(NA, NA)	$\Rightarrow$ `E`

Figure 13: Normalized dispatch rules for running example.

2. eJSTK discards the half-normalized dispatch rules whose conditions specify combinations of operand datatypes that are never given to the instruction in accordance with the operand specifications.
3. eJSTK further decomposes each half-normalized dispatch rules by decomposing VM-datatypes into VM-reptypes.

The resulting collection of normalized dispatch rules for the running example is shown in Fig. 13. Note that this running example assumes that the operand specifications require the instruction to accept any operand datatypes.

7.2.2. Step 2: Decision tree construction

eJSTK constructs a decision tree in which each normalized dispatch rule created in Step 1 corresponds to a path from the root to a leaf. An internal node of the tree examines a pointer tag value or a header tag value of some operand. An internal node has edges, each of which is labeled with a pointer tag value or a header tag value to a child. A leaf of the tree is a C code fragment. Because the decision tree is made from the normalized dispatch rules created in Step 1, a node does not necessarily have edges for all the tag values; an edge for a tag value of an unspecified datatype in the operand datatypes is not created. Formally speaking, the result of the decision tree is undefined if the value examined by a node is not found in any of the labels of the edges. However, such a case never happens as long as the operand specifications specify all the possible operand datatypes, which we assume.

The constructed decision tree is *ordered*; i.e., the pointer tags and header tags are examined in the same order along all paths from the root to the leaves. The ordered decision tree examines the pointer tags of all operands from the first operand to the last and then examines the header tags, if any, of all operands, also from the first operand to the last.

Furthermore, along every path from the root to a leaf, both the pointer tag value and header tag value of every operand are examined. This is done to simplify the optimization algorithm used in Step 3. If the VM-reptype of an operand has a unique pointer tag value, the datatype of the operand can be judged only by the pointer tag value, as mentioned in Sect. 2.2. Nevertheless, eJSTK always creates internal nodes in the ordered decision tree for the header tags of such VM-reptypes. In particular, even if a tagged pointer contains an immediate value, such as normal_fixnum in the running example, and the

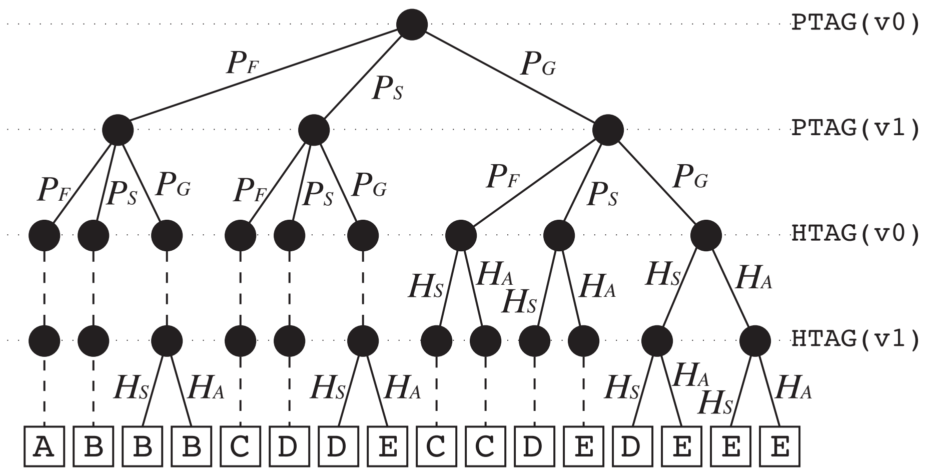


Figure 14: Decision tree for running example. Edges drawn with dashed lines indicate that VM-reptype does not have header tag.

operand has no header tag, eJSTK creates internal nodes for the non-existent header tags.

The decision tree for the running example is shown in Fig. 14. The root node tests the pointer tag of operand `v0`. There are three possibilities: P_F (for normal_fixnum), P_S (for direct_string), and P_G (for normal_string and normal_array). Next, the second level node tests the pointer tag value of `v1`. The third level node tests the header tag of operand `v0`. If the pointer tag of operand `v0` is P_F or P_S, the VM-reptype is normal_fixnum or direct_string, which does not have a header tag. For such cases, the decision tree yields a child node without testing the tag. Edges to these children are drawn with dashed lines in Fig. 14.

It is worth noting that an internal node created for the header tag of a VM-reptype with a unique pointer tag value has exactly one child. This property is important for efficiency because eJSTK removes such nodes in Step 3.

Proposition 7.1. *In the constructed ordered decision tree, if an internal node examines the header tag of an operand of a VM-reptype with a unique pointer tag value, this internal node has exactly one child.*

7.2.3. Step 3: Decision tree optimization

eJSTK transforms the ordered decision tree into a DAG to produce more efficient and compact type-based dispatching code.

For a formal discussion, we define an ordered decision diagram (ODD). A decision diagram is a rooted DAG. A terminal node is called a *leaf*. A leaf is associated with a C code fragment. When a leaf L is associated with a C code fragment C, we denote $code(L) = C$. A non-terminal node is called a *decision node*. A decision node is associated with a test that examines the pointer tag value or the header tag value of an operand. We denote the tag value of the operand that a decision node V examines as $\mathit{test}(V)$. A decision node with a single successor is called a *branchless* node, and one with multiple successors is called a *branching* node. An edge from a decision node is labeled with one or more pointer tag values or header tag values. An ODD is a decision diagram in which tests appear in the same order along all paths from the root to the leaves.

We denote a decision node of an ODD by using a partial function V that maps a pointer tag value or a header tag value t to $V(t)$, the successor node connected through the edge labeled with t. If such a successor does not exist, $V(t)$ is undefined. If a decision node V is for a non-existent header tag introduced in Step 2, V has exactly a single successor node U. In this case, we define that, for every header tag value t, $V(t) = U$.

The input of an ODD is a combination of operands. For example, $\langle \mathtt{v0}, \mathtt{v1} \rangle$ is given as input to an ODD for a two-operand instruction. The output of an ODD is a C code fragment. We denote that an ODD with root R yields a C code fragment C for a combination of operands I as $\mathit{dispatch}(R, I) = C$. We consider not only the entire ODD but also a sub-ODD, which is some node V (the root of the sub-ODD in the ODD) and its transitive closure. In this case, we denote $\mathit{dispatch}(V, I) = C$ if the sub-ODD starting from V yields C for input I. It should be noted that $\mathit{dispatch}(V, I)$ may not be defined for some input I, which we denote $\mathit{dispatch}(V, I) = \bot$, because the datatypes of operands might be limited in accordance with the operand specifications.

Before proceeding to the optimization process in Step 3, we define the *equivalence* and *upward-compatibility* of ODDs and the *consistency* of nodes.

Definition 7.1. *Two ODDs starting from nodes U and V are* equivalent *iff, for every combination of operands I, the outputs of both ODDs are defined and $\mathit{dispatch}(U, I) = \mathit{dispatch}(V, I)$ or neither is defined.*

Definition 7.2. *An ODD starting from node V is* upward-compatible *with that from U, or simply V is upward-compatible with U, iff $\mathit{dispatch}(U, I) = \mathit{dispatch}(V, I)$ for every combination of operands I such that $\mathit{dispatch}(U, I)$ is defined.*

Please note that we do not care about $\mathit{dispatch}(V, I)$ if $\mathit{dispatch}(U, I) = \bot$ in Definition 7.2.

Definition 7.3. *Two nodes U and V in an ODD are* consistent *iff $\mathit{test}(U) = \mathit{test}(V)$ and one of the following conditions is satisfied.*

1. *Both U and V are leaves and $\mathit{code}(U) = \mathit{code}(V)$.*
2. *Both U and V are branching nodes that satisfy that, for every pointer tag value or header tag value t such that both $U(t)$ and $V(t)$ are defined, $U(t) = V(t)$.*
3. *Both U and V are branchless nodes, and $U(s) = V(t)$, where s and t are the labels of their only edges.*

For example, in Fig. 15, in which we assume $\mathit{test}(A) = \mathit{test}(B)$ for decision nodes A and B, A and B are consistent in cases (a) and (b) because they

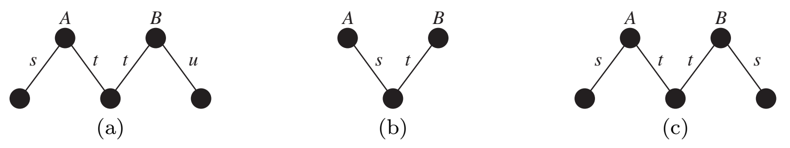


Figure 15: Consistency; A and B are consistent in (a) and (b), but not in (c). We assume $test(A) = test(B)$.

satisfy the second and third conditions, respectively. In case (c), A and B are not consistent because they are branching nodes that are defined on tag value s, but $A(s) \neq B(s)$. Please note that U and V are not consistent if one is a branching node and the other is branchless.

The input for Step 3 is the ordered decision tree constructed in Step 2. Since an ordered decision tree is a special case of ODD, we can rephrase this: the input for Step 3 is the ODD constructed in Step 2. In addition, the input decision diagram has a property that, every path examines all the tag values. We call ODD with this property a *fully-testing* ODD. Let the *depth* of V, denoted as $depth(V)$, be the distance from the root to a node V.

Proposition 7.2. *In a fully-testing ODD, for two nodes U and V, $depth(U) = depth(V)$ if and only if $test(U) = test(V)$.*

The output of Step 3 is an optimized ODD that is equivalent to the input ODD. We next describe the optimization algorithm and then the properties of an optimized ODD.

Optimization algorithm. To generate the optimized ODD, eJSTK first combines consistent nodes, and then it removes branchless nodes.

If two nodes are consistent, their "combined" node can be defined.

Definition 7.4. *Let U and V be consistent nodes. Their* combined node*, denoted as $U \oplus V$, is defined as follows.*

- *If both U and V are leaves, $U \oplus V = U$.*
- *If both U and V are decision nodes,*

$$(U \oplus V)(t) = \begin{cases} U(t) & \text{if } U(t) \text{ is defined.} \\ V(t) & \text{otherwise} \end{cases}$$

The combined node of U and V, $U \oplus V$, is upward-compatible with both U and V. Thus, eJSTK replaces both U and V with $U \oplus V$. Or, we say simply that eJSTK *combines* U and V.

Non-consistent nodes U and V may become consistent after combining their successor nodes. For example, if both U and V are branching nodes and $U(t)$ and $V(t)$ are combined for all t such that both $U(t)$ and $V(t)$ are defined, U and V become consistent. Thus, eJSTK repeatedly combines two consistent nodes from the leaves to the root to maximize the opportunity for combining.

Because the input ODD is a fully-testing one, two nodes U and V are consistent only if $\mathit{depth}(U) = \mathit{depth}(V)$ by Definition 7.3 and Proposition 7.2. This property is preserved even after a pair of nodes is combined. Therefore, once we combine all the consistent pairs of nodes in some distance N from the root or farther, any node V such that $\mathit{depth}(V) \geq N$ never becomes consistent with other nodes even if we combine nodes closer to the root.

Algorithm 7.1. *First combine every consistent pair of leaves until no consistent pairs of leaves are left. Then, combine every consistent pair of decision nodes whose successors are leaves until no such consistent pairs are left. In this way, from the leaves to the root, combine every consistent pair of nodes whose depths are the same. until no consistent pairs in the depth are left.*

The resulting ODD is equivalent to the original ODD. In addition, type-based dispatching code generated from the resulting ODD should be more compact because each leaf is translated into the associated C code fragment, and each decision node is translated into a `switch-case` statement in Step 4.

eJSTK next removes all branchless nodes.

Algorithm 7.2. *Remove all branchless nodes. Whenever a branchless node V with predecessor node P and successor node S is removed, replace the destinations of all edges from P to V with S. The new predecessor node P' is defined as*

$$P'(t) = \begin{cases} S & \text{if } P(t) = V. \\ P(t) & \text{otherwise} \end{cases}$$

In theory, Algorithm 7.2 can create a new consistent pair of nodes. Suppose that two branching nodes U and V have edges labeled with t to branchless successor nodes $U(t)$ and $V(t)$ before performing Algorithm 7.2. Suppose further that $U(t)$ and $V(t)$ share a successor W through edges with different labels, that is, $U(t)(a) = V(t)(b) = W$ for some a and b ($a \neq b$). Suppose even further that U and V would be consistent if $U(t)$ and $V(t)$ were consistent, that is, for every label s such that $s \neq t$ and both $U(s)$ and $V(s)$ are defined, $U(s)$ and $V(t)$ are consistent. In this case, Algorithm 7.2 removes $U(t)$ and $V(t)$, and we have $U(t) = V(t) = W$. Thus, U and V become consistent by Algorithm 7.2. However, we did not find such cases in our experimentation using our instruction definition.

After combining all consistent nodes using Algorithm 7.1, our running example is transformed into the ODD shown in Fig. 16. Combining decision nodes can result in an edge having multiple labels. For example, the edge above `B` has labels H_S and H_A.

After removing all branchless nodes using Algorithm 7.2, our running example is transformed as shown in Fig. 17.

Properties of optimized ODD. We first define a *redundant* decision node. Intuitively, if an ODD has a redundant decision node, generated type-based dispatching code tests the header tag of an operand of a VM-reptype with a unique pointer tag value.

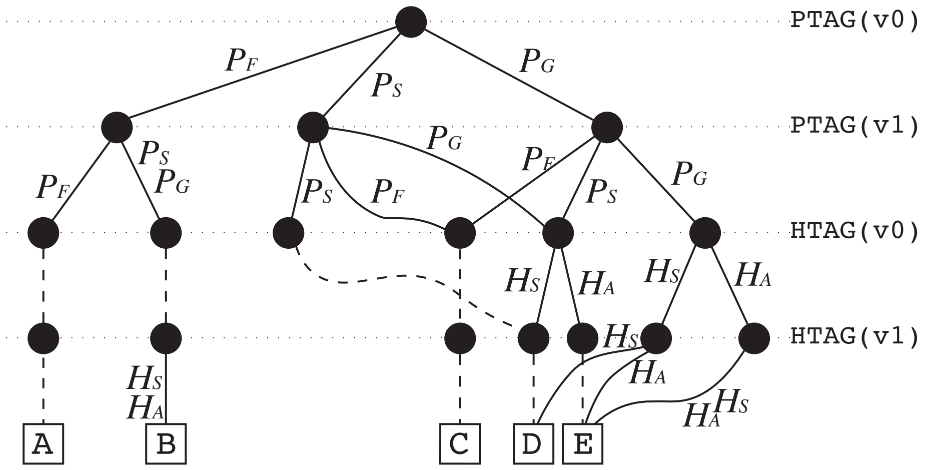


Figure 16: ODD for running example after combining consistent nodes.

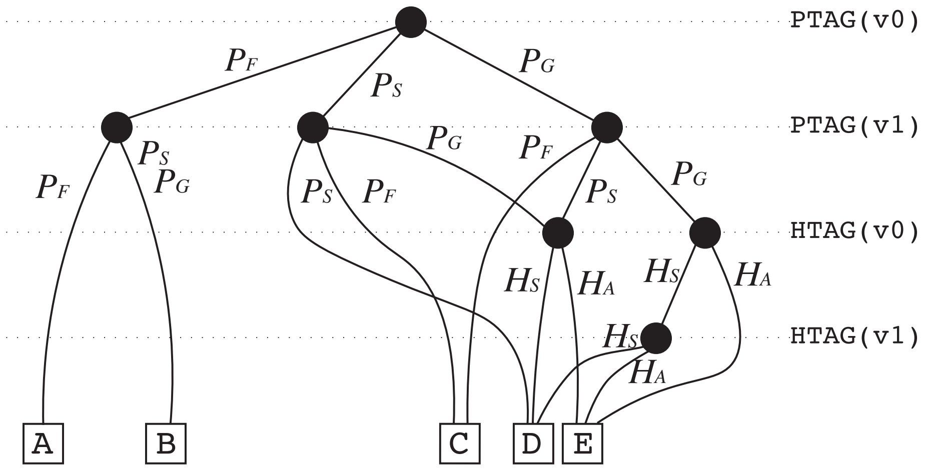


Figure 17: ODD for running example after removing branchless decision nodes.

Definition 7.5. *A decision node V is* redundant *in an ODD iff it examines the header tag of an operand of a VM-reptype with a unique pointer tag value and the pointer tag is examined by another decision node along the path from the root to V.*

The ODD optimized using Algorithms 7.1 and 7.2 has two properties.

1. There are no branchless decision nodes.
2. There are no redundant decision nodes.

Property 1 is obviously satisfied because Algorithm 7.2 removes all branchless nodes.

Regarding property 2, Proposition 7.1 guarantees that the ODD input to Algorithm 7.1 has the following weaker property:

$2'$. There are no redundant *branching* nodes.

Proposition 7.3 states that Algorithm 7.1 preserves property $2'$. Therefore, the optimized ODD has property 2 because Algorithm 7.2 removes all branchless nodes.

Proposition 7.3. *For every combining operation performed using Algorithm 7.1, if the ODD before the combining operation does not have a redundant branching node, the resulting ODD also does not have a redundant branching node.*

Proof. Assume that the combining operation replaces consistent nodes U and V with $U \oplus V$.

- If U and V are leaves, $U \oplus V$ is also a leaf.
- If U and V are branching nodes, neither U nor V is redundant because the ODD before combining does not have a redundant branching node by the premise of this proposition. Thus, $U \oplus V$ is also not redundant.
- If U and V are branchless decision nodes, their successor nodes are the same node. Thus, $U \oplus V$ is also a branchless node.

Therefore, the resulting ODD does not have a redundant branching node. □

Theorem 7.1. *The optimized ODD does not have redundant decision nodes.*

Proof. At the beginning of Step 3, every decision node examining a header tag of an operand of a VM-reptype with a unique pointer tag value is branchless. Thus, the ODD input to Algorithm 7.1 does not have redundant branching nodes by definition of redundancy. By Proposition 7.3, the ODD output by Algorithm 7.1 does not have redundant branching nodes. Because Algorithm 7.2 removes all branchless nodes, all decision nodes remaining in the optimized ODD are branching nodes. Therefore, all decision nodes are not redundant. □

```
switch(PTAG(v1)) {
case T_FIXNUM:
     switch(PTAG(v2)) {
     case T_FIXNUM:  { A } break;
     case T_GENERIC: case T_DSTRING: { B } break;
     } break;
case T_DSTRING:
     switch(PTAG(v2)) {
     case T_FIXNUM:  goto LC;
     case T_DSTRING: goto LD;
     case T_GENERIC: goto L1;
     } break;
case T_GENERIC:
    switch(PTAG(v2)) {
    case T_FIXNUM:  LC: { C } break;
    case T_DSTRING:
        switch(HTAG(v1)) {
        case HTAG_STRING: LD: { D } break;
        case HTAG_ARRAY:  LE: { E } break;
        } break;
    case T_GENERIC:
        switch(HTAG(v1)) {
        case HTAG_STRING:
            L1:switch(HTAG(v2)) {
            case HTAG_STRING: goto LD;
            case HTAG_ARRAY:  goto LE;
            } break;
        case HTAG_ARRAY: goto LE;
        } break;
    } break;
}
```

Figure 18: Generated C code for running example.

7.2.4. Step 4: C code generation

eJSTK generates C code for type-based dispatching from the optimized ODD. This is a straightforward process using a depth-first search over the ODD. A decision node is translated into a `switch-case` statement that branches on the basis of the associated test with the node, and a leaf is translated into its associated C code fragment. If a node is shared with multiple predecessor nodes, eJSTK generates the code for the node when it visits the node for the first time and emits a `goto` statement for the second time and later.

The generated C code for our running example is shown Fig. 18.

8. Evaluation

We evaluated the effectiveness of eJSTK from three viewpoints: (1) effectiveness of customization, (2) effectiveness of optimization in the instruction part generator, and (3) comparison of generated VMs with other JavaScript engines.

Execution environment. We used two environments, ARM and x86. Though x86 is not an embedded system, we used it to show the potential of eJSTK.

ARM: A Raspberry Pi 3 with the BCM2837 64-bit CPU (1.2 GHz), the Raspbian GNU/Linux 8 (Linux kernel 4.9.35) OS, and the GCC 7.3.0 compiler.

x86: A desktop computer with the Intel(R) Core(TM) i7-6700K CPU (4.00 GHz), the Ubuntu 14.04.5 (Linux kernel 3.13.0) OS, and the GCC 7.3.0 compiler.

We used the `-Os` compilation option to reduce the code size for both environments.

For all executions, we used a 10-MB heap to minimize the effect of garbage collection. The current eJSVM use a naive mark-sweep garbage collection that is not well tuned. Furthermore, our object model was not specialized for embedded systems, and thus required a large heap. The garbage collection and object model are beyond the scope of this work, and are left for future work.

Benchmark programs. We selected a set of benchmark programs from SunSpider Benchmarks[9] version 0.9 and made minor modifications so that they could run on our eJSVM. The selected benchmarks are listed in Fig. 19.

The benchmark names start with their category names, which represent their characteristics:

- Benchmarks with 3d- and math- perform arithmetic operations on floating-point numbers.
- Those with access- access properties of objects or arrays intensively.
- Those with bitops- perform bitwise operations on integers that fit our fixnumVM-datatype.
- Those with string- deal with many strings.

The modifications we made were in

- reduction in the number of iterations, and
- rewriting so as not to use some built-in functions; e.g., we replaced `Math.random`, which was used in string-base64 to generate the text to be encoded, with a manually written pseudo-random number generator function.

VMs. We compared four VMs generated with eJSTK and a manually tuned `handcrafted` VM in terms of their binary size and elapsed time in executing the benchmark programs.

We made two datatype specifications and two operand specifications and combined them to generate the four VMs.

- The `default` datatype specifications were the default settings shown in Table 2.
- The `arraytag` datatype specifications correspond to Example 1 in Sect. 4.3. A unique pointer tag value was assigned to normal_array. The value was 001, which was not used in the default setting.

[9]https://webkit.org/perf/sunspider/sunspider.html

- The `any` operand specifications were used to generate instructions that supported any operand datatype.
- The `fixnum` operand specifications were used to generate arithmetic and relational instructions that accepted only fixnums. The add instruction is an exception; it accepted either two fixnums or two strings. When operands with unsupported datatypes were given, the VM possibly crashes. More precisely, it executes code for other datatypes[10].

We denote the four VMs by concatenating the names of the specifications: `default-any`, `arraytag-any`, `default-fixnum`, and `arraytag-fixnum`.

The `handcrafted` VM had the same datatype representations and accepted the same combinations of operand datatypes for each instruction as a VM with the default setting, but we manually wrote all of the VM code and tuned it as much as possible. For example, its instructions dealt with only probable cases in the interpreter loop and called slow path functions for other cases; e.g., the add instruction called the slow path function unless both of its source operands were numbers or both were strings. This may have contributed to keeping the main loop of the interpreter small.

Note that we did not implement any specialized representation of strings, such as direct_string, which we used as an example in Section 4.2 and Section 6.

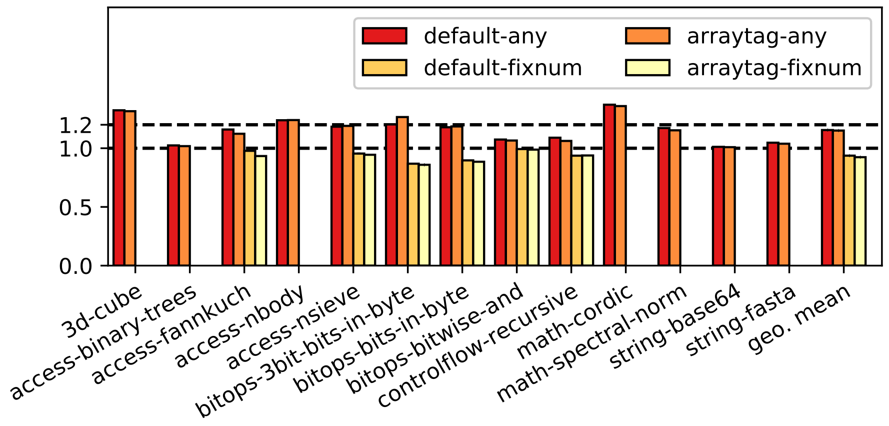
8.1. Effectiveness of customization

Figure 19 presents the elapsed times for executing the benchmarks in the ARM and x86 environments. All results were normalized to the elapsed times of the `handcrafted` VM.

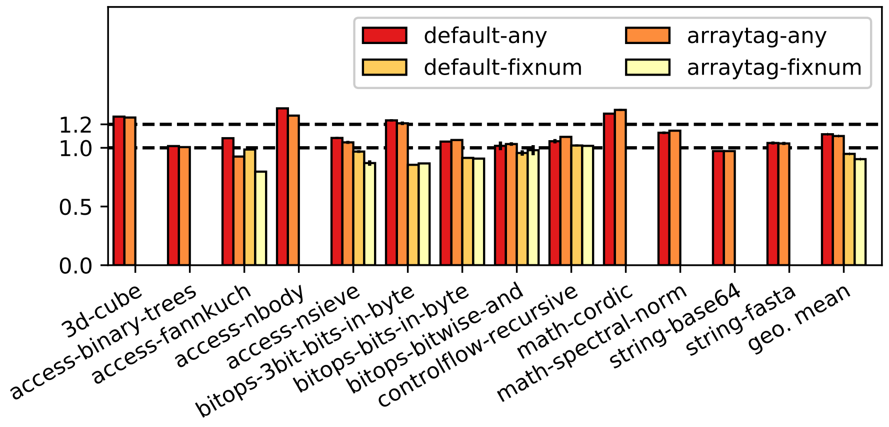
Figure 20 compares the sizes of the VMs. The bars in the group labeled optimized represent the sizes of VMs generated using the decision tree optimization described in Sect. 7.2. The colored areas represent the sizes of the main interpreter loop (and out-of-line slow path functions for `handcrafted`), and the grey areas represent the sizes of the remaining part excluding the built-in functions. We excluded the sizes for built-in functions because they are assumed to be selectively included in the VMs.

Before examining the effectiveness of the customization, we examine the overall performance of the code generator. The `default-any` VM had the same datatype representations as the `handcrafted` VM, and its instructions accepted any operand datatypes. Comparing these two, the manually tuned `handcrafted` VM was around 1.2 times faster or more for 7 out of 13 benchmarks on the ARM environment. We think the manual tuning applied to the `handcrafted` VM worked effectively for these benchmarks. For example, in the `handcrafted` VM, we chose operand datatypes that are expected to be used frequently for each instructions and inlined VM internal functions on such hot paths. More specifically, we inlined `to_double` VM internal function, which converts a value of any

[10] eJSTK can also generate VMs that terminate gracefully when unsupported datatypes are given. Such VMs are larger than those that crash.

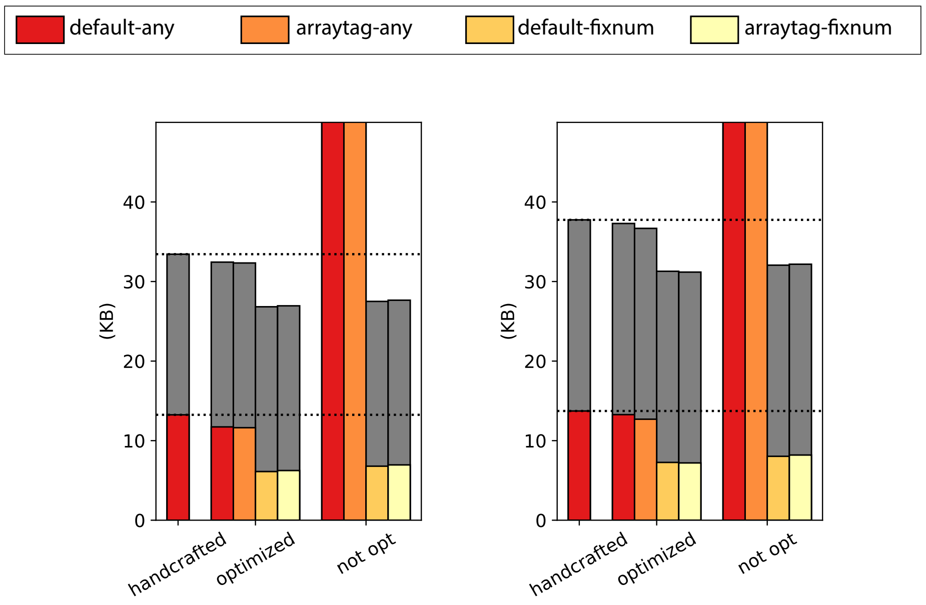


(a) ARM



(b) x86

Figure 19: Elapsed times normalized to `handcrafted`. Missing data indicates that execution failed.



(a) ARM: the sizes of `default-any` and `arraytag-any` with no optimizations were 151 KB and 163 KB.

(b) x86: the sizes of `default-any` and `arraytag-any` with no optimizations were 158 KB and 181 KB.

Figure 20: Sizes of generated and `handcrafted` VMs.

datatype into a `double` value of the C language, for `fixnum` and `flonum` operands of arithmetic and relational instructions. The `to_double` function performs further dispatching to call a datatype-specific type conversion function, such as `fixnum_to_double`. On the inlined paths, the interpreter directly calls these datatype specific converting functions. The effect of this inlining contributed to two thirds of the difference of elapsed times between the `handcrafted` and `default-any` VMs for 3d-cube on ARM. Concretely speaking, the absolute times for 3d-cube on ARM were 3.60 seconds for the `handcrafted` VM, 4.77 seconds for the `default-any` VM, and 4.37 seconds for the `handcrafted` VM without inlining tuning.

As for the VM sizes, the `default-any` VM was smaller, but not by much. This shows that the manual tuning caused overhead in size. We conclude that the speed and size of the generated VM were comparable to those of the VM programmed manually in the traditional way.

Let's turn now to the effectiveness of customization. The purpose of the customization was to generate an appropriate VM for each program, each demand such as execution time and VM size, and each condition such as execution environment and available bits in pointers for unique pointer tag values. Thus, if the best customized VM differs between benchmark programs, demands, or conditions, we can say that customization was effective.

Comparing the `default-any` and `default-fixnum` VMs, `default-fixnum` was faster and smaller for all benchmarks that `default-fixnum` could execute. Comparison of the `arraytag-any` and `arraytag-fixnum` VMs showed the same tendency. Thus, we can conclude that limiting operand datatypes is effective. Although the VM cannot execute programs that need instructions to accept more operand datatypes than specified in the operand specifications, this is not a problem because a customized VM is specialized for a specific program to be run on the VM.

Next, we compare the `default-any` and `arraytag-any` VMs to examine the effects of customizing datatype representations. The `arraytag` datatype representations were specialized for programs that intensively performed property access (including array access with the fixnum index). Thus, we assigned a unique pointer tag to normal_array.

In contrast to our expectation, `default-any` and `arraytag-any` showed similar performance both in execution speed and VM size. The only exception was access-fannkuch on the x86 environment. However, when we used the `arraytag` datatype specifications together with the `fixnum` operand specifications, `arraytag-fixnum` outperformed `default-fixnum` for access-sieve as well as access-fuannkuch. These benchmarks accessed arrays intensively. For the other benchmarks, `arraytag-fixnum` was as fast as `default-fixnum` as long as both could run. As for size, `arraytag-fixnum` was not worse than `default-fixnum`. Since `arraytag-fixnum` required more bit patterns for pointer tag values to represent normal_array with a unique pointer tag value, this result shows that we succeeded in generating an appropriate VM for each condition on the number of available bits for pointer tag by customization. Therefore, we conclude that customization of datatype representations was effective.

8.2. Effectiveness of optimization

In this section, we compare optimized VMs (optimized) with not optimized ones (not opt) to examine the effectiveness of the decision tree optimization described in Step 3 of Sect. 7.2.

The sizes of the VMs are shown in Fig. 20. Figures 21 and 22 show the elapsed times in executing each benchmark. They were normalized to those of the `handcrafted` VM.

Figure 20 indicates that the decision tree optimization is effective for reducing VM size. Without the decision tree optimization (not opt), the generated VMs were larger than the `handcrafted` VM. In particular, the VMs with `any` operand specifications were about 10 times larger than the `handcrafted` VM. This was because `any` operand specifications required more complicated type-based dispatching than `fixnum` operand specifications. Figures 21 and 22 show that the optimization made the VMs substantially faster.

Code generation took about 9 seconds in total for all 29 instructions regardless of whether the optimization was enabled.

We conclude that the optimizations were effective in terms of both speed and size.

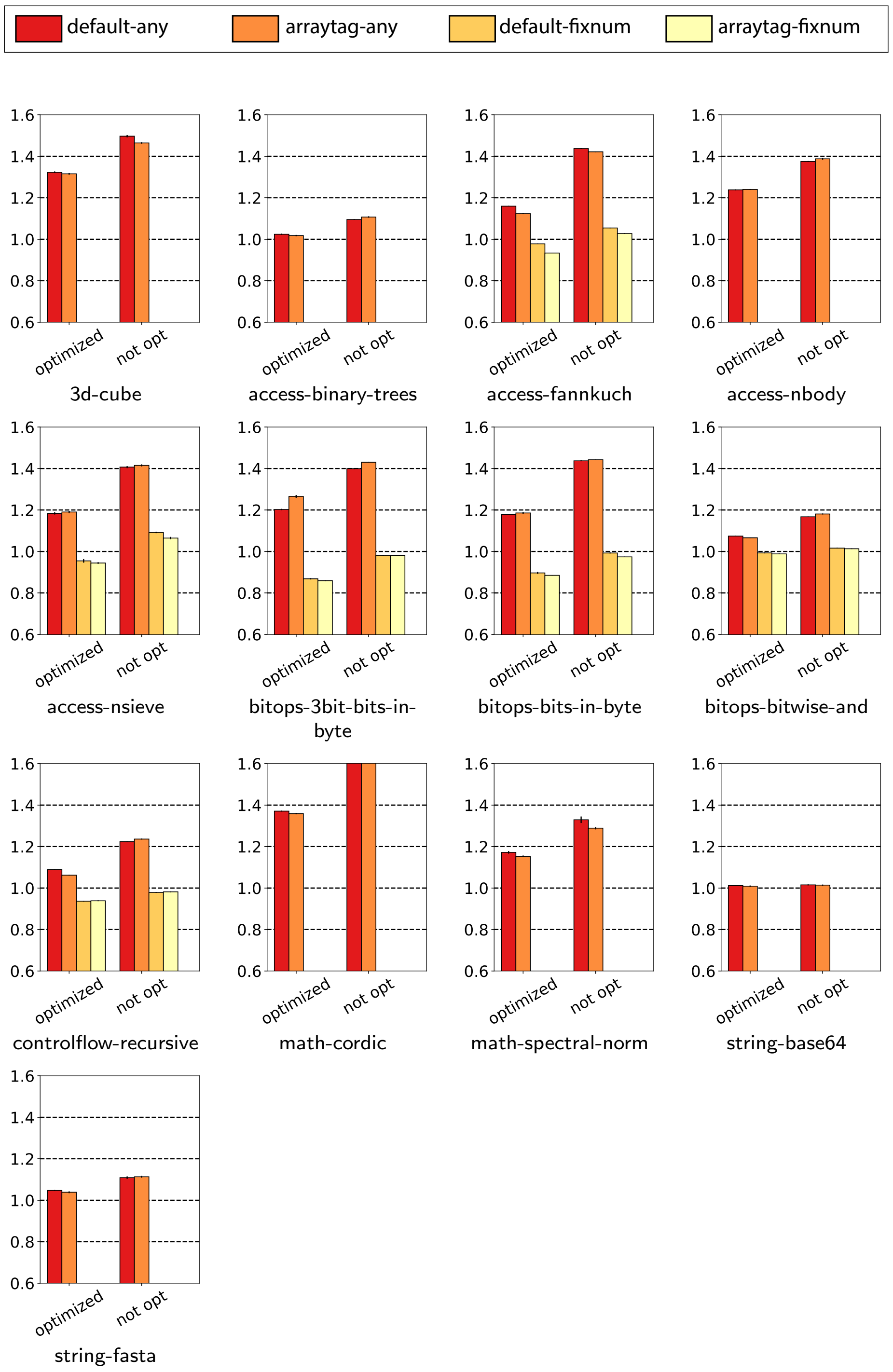


Figure 21: Elapsed times on ARM normalized to `handcrafted`.

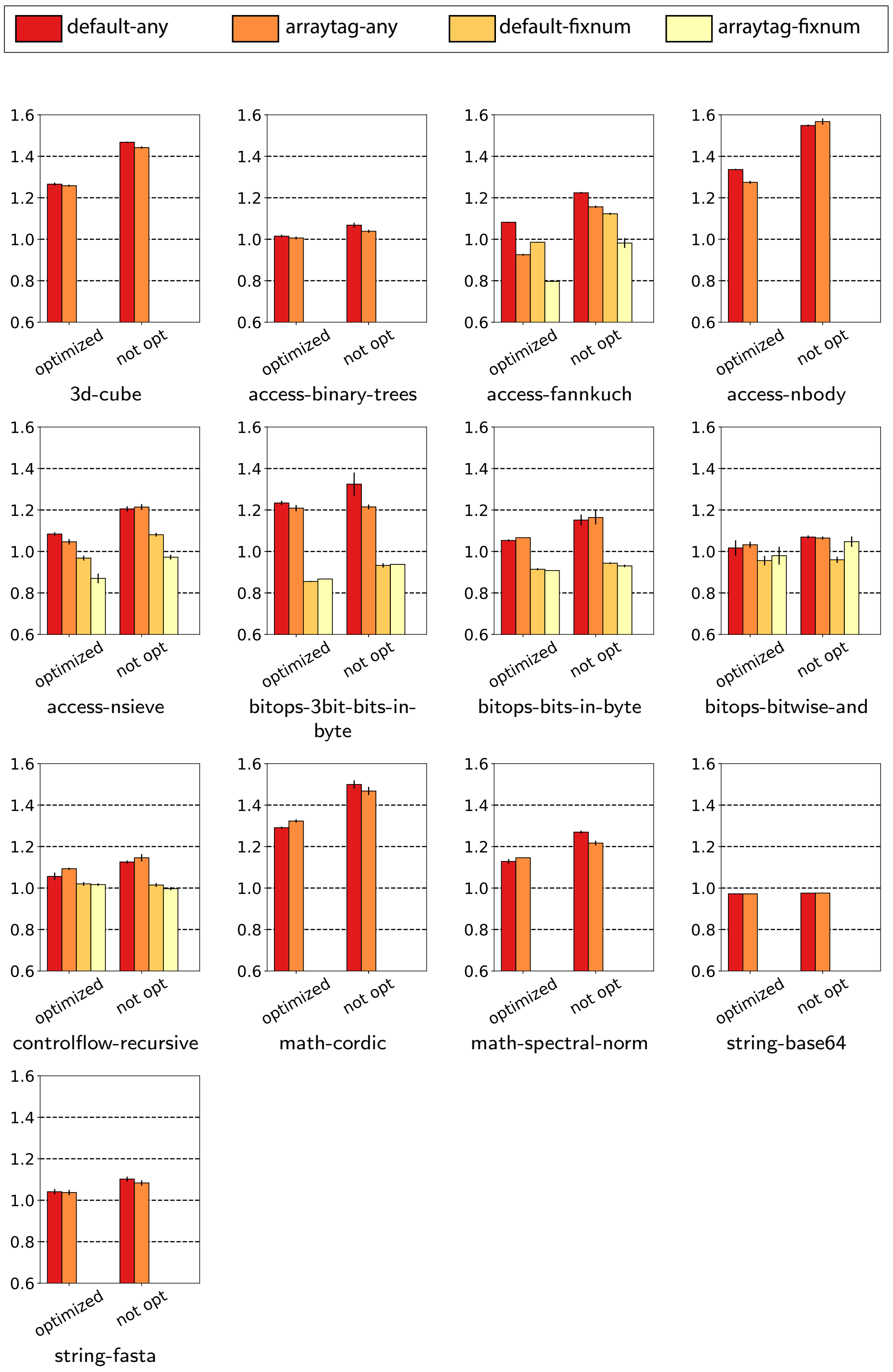


Figure 22: Elapsed times on x86 normalized to `handcrafted`.

Table 3: Comparison with other JavaScript engines in the ARM environment. Elapsed time including garbage collection time (in parentheses) for each benchmark (sec).

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

8.3. Comparison with other JavaScript engines

We compared the `default-any` VM with other JavaScript engines for embedded systems, namely JerryScript (Version 1.0) and V7 (Version 3.0) in the ARM environment. Both engines were run with their default settings. From their default settings, the heap size of JerryScript was 512 KB fixed, and that of V7 grew on-demand, starting at 22 KB.

Table 3 shows the elapsed time and garbage collection time for each benchmark. Every elapsed time includes garbage collection time. For JerryScript, only elapsed times are shown because JerryScript uses reference-counting garbage collection, which prevented us from measuring garbage collection times.

In comparison to JerryScript, the results for eJSVM were comparable; eJSVM was faster for 8 out of 13 benchmarks. The string-base64 benchmark was an exception; it was much slower on eJSVM because eJSVM could not create strings efficiently.

V7 was much slower than eJSVM even if we compare pure computation times except garbage collection. We think that this is due to V7's implementation of property accesses including array index accesses, which uses association lists.

eJSVM used a much larger heap to reduce the effect of our poor memory management. Nevertheless, eJSVM and JerryScript were comparable even if we assume that half of the execution times in JerryScript were spent on garbage collection. In addition, eJSVM was faster than V7 in pure computation times. Thus, the larger heap setting did not over-compensate for the memory management implementation.

To sum up, these results indicate that eJSVM has comparable performance to that of other JavaScript engines for embedded systems.

We also compared eJSVM in the x86 environment with more mature JavaScript VMs for desktop computers, Rhino (Version 1.7.9), SpiderMonkey (Version C24.2.0), and V8 (Version 7.0.276.28-node.5), under their default settings. For Rhino, the Java heap size was initially 118 MB, and it grew up to 207 MB. For

Table 4: Comparison with other JavaScript engines in the x86 environment. Elapsed time including garbage collection time (in parentheses) for each benchmark (sec).

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

V8, the heap size was initially 4.2 MB, and it grew up to 11.2 MB. We could not find the heap sizes for SpiderMonkey.

Table 4 presents the results. Every elapsed time includes garbage collection time.

The results of eJSVM were comparable with Rhino, but almost one order magnitude slower than SpiderMonkey and V8. Though the target of eJSVM is low-memory systems, performance improvement of customized eJSVM is our future work.

9. Related Work

Vmgen [8, 9] is a virtual machine interpreter generator that has special support for stack machines. Vmgen generates C code for executing VM instructions by taking as input VM instruction descriptions. Although both Vmgen and eJSTK generate VM interpreters, there are two major differences between the two. First, their directions for customization are totally different; Vmgen is "customizable" in the sense that Vmgen enables the programmer to define required instructions and generate VM code for various languages, e.g., Gforth [10] and Cacao JVM [11]. In contrast, eJSTK's target language is fixed to JavaScript, and its customizability is based on the internal structures of datatypes to make the generated VM adapt to the target application. To sum up, Vmgen is aimed at instruction-centric customization, while eJSTK is aimed at datatype-centric customization. Second, the features of target languages are different. An instruction specification in Vmgen contains the type information of the items popped from and pushed onto the stack. For example, the `iadd` instruction of Java VM is defined as taking two integers from the stack and pushing an integer onto the stack. This means that datatypes concerning each instruction are supposed to be fixed in its target language. Compared to this, eJSTK's target is JavaScript and datatypes of values given to an instruction are therefore dynamically determined at runtime. Thus, it is necessary for each instruction to perform type-based dispatching efficiently.

Beatty et al. [12] presented an efficient Java interpreter for connected devices and embedded systems. They used Vmgen for interpreter generation. A Java application bytecode was translated into direct threaded code during which a number of optimizations were applied. They focused on reducing the cost of instruction dispatching, while eJSTK focused on reducing the cost of type-based dispatching in the execution of each instruction.

Mruby [13, 14] is a lightweight implementation of Ruby. It is intended to be linked with target applications, which are typically those on embedded systems. Both mruby and eJSTK share the same purpose but take different approaches. While mruby's approach makes it possible for the programmer to select required modules, ours makes it possible to select required datatypes and their representations.

Many studies have been done on the dynamic customization or optimization of language interpreters. Quickening [15, 16] and superinstruction [17, 18] are well-known optimization techniques based on runtime instruction rewriting. Quickening replaces an instruction with one that is specialized by using runtime information, typically by its operands. Superinstruction replaces a successive sequence of more than one instruction with an equivalent, i.e., merged single instruction. We think that these techniques can be applied to a customized eJSVM. It may be possible to automatically generate a specialized/merged instruction from the instruction definitions without much effort because an instruction's behavior is described according to every individual combination of given datatypes.

NaN boxing [19] is an implementation technique for value representation. This technique embeds an entire pointer into the mantissa part of a Not-a-Number representation of the IEEE Standard for Floating Point Arithmetic. By using Nan boxing, it is not necessary to allocate a data structure for a floating point number within the heap area. It might be better to allow the programmer to use this technique for more thorough customization. It is left for our future work.

Kim et al. [20] introduced Typed Architectures that achieve efficient program execution of scripting languages from the observation that dynamic types, i.e., type checking and method dispatch, are main causes of inefficiency of scripting languages. Typed Architecture lets each internal representation of a value retain high-level type information at an instruction set architecture level and performs type checking implicitly with the pipeline with instruction execution. Their work and ours concern the same problem of scripting languages, but the approaches taken are totally different; they relied on hardware-based acceleration of type checking, while our work addressed this problem by means of providing the ability of VM customization by the programmer.

Würthinger et al. [21] and Humer et al. [22] presented an idea for implementing abstract syntax tree (AST) interpreters where AST was appropriately modified during program execution. They took a layered approach: a guest VM (written in Java) was on top of a host VM (written in C++). The AST interpreter rewrote the tree by replacing one node with another that was specialized that could more quickly carry out expected operations. This might be simpler than the instruction rewriting previously described, but it would be difficult to apply this idea to embedded systems due to their limited resources.

This work is also related to *interpreter specialization* [23, 24], which applies the program specialization technique to an interpreter to generate an efficient implementation. In a sense, eJSTK can be regarded as a special kind of a compile-time interpreter specialization approach. The program generation by eJSTK is not a pure source-to-source (C-to-C) program transformation but a DSL-to-C program transformation to adapt eJSVM for each target application. This DSL-to-C transformation enables eJSTK to generate compact and efficient type-based dispatching code by means of decision diagrams.

ODD and its optimization process described in Sect. 7.2 are related to ordered binary decision diagram (OBDD) and normalized OBDD (NOBDD) [25]. There are two main differences between an ODD in this paper and an OBDD. First, the number of branches from a decision node in our ODD is not limited to two; it might be one, i.e., branchless, or more than two, depending on the tag value assignments described in the datatype specifications and the number of possible datatypes for the operand that corresponds to the decision node. Second, a decision node in our ODD is represented as a *partial* function, while that of OBDD is represented as a total function whose domain is $\{0, 1\}$. Thus, the result of the function for a decision node is undefined, or "don't care", for an impossible pointer tag / header tag value. The optimization process in Sect. 7.2 uses this fact in combining two nodes to form an upward-compatible node with the two.

Code generation on the basis of ODD is related to multiple dispatching in object-oriented systems, which is the selection of methods (i.e., multi-methods / open-methods) to be invoked based on the dynamic types of more than one argument. One major technique for dispatching of a multi-method invocation uses a dispatch table, which holds dispatching entries for all possible combinations of the arguments' types. This dispatch table technique can be optimized by compressing the table to eliminate the number of dispatching entries [26]. Comparing our ODD-based technique with the dispatch table technique is left for our future work.

10. Conclusion

We presented a novel approach to generating customized JavaScript VM interpreters. The proposed framework, eJSTK, is datatype-centric in the sense that customizations with respect to the representations of datatypes specified by the programmer are applied. A generated VM interpreter has efficient type-based dispatching code according to the datatype specifications.

We would like to explore the selection of VM instructions in a customized eJSVM in future work. We can know the set of VM instructions used by a program from the compiled program to be run on a customized VM. Thus, we are able to remove such instructions that are never used from the main loop of the VM. We are also considering to generate both operand specifications and datatype specifications automatically from an application. Such a system would help the programmer develop applications for embedded systems by using eJSTK.

The eJSTK is available on GitHub `https://github.com/plasklab/ejstk`.

Acknowledgement

The authors would like to thank all members involved in the eJSVM project. This work was partly supported by the JSPS KAKENHI Grant Number 16K00103.

References

[1] P. Koopman, Better Embedded System Software, Drumnadrochit Press, 2010.
URL `http://koopman.us/book.html`

[2] G. Chadha, S. Mahlke, S. Narayanasamy, Efetch: Optimizing instruction fetch for event-driven web applications, in: Proc. 23rd International Conference on Parallel Architectures and Compilation, PACT 2014, 2014, pp. 75–86. `doi:10.1145/2628071.2628103`.

[3] Y. Zhu, D. Richins, M. Halpern, V. J. Reddi, Microarchitectural implications of event-driven server-side web applications, in: Proc. 48th International Symposium on Microarchitecture, MICRO-48, ACM, 2015, pp. 762–774. `doi:10.1145/2830772.2830792`.
URL `http://doi.acm.org/10.1145/2830772.2830792`

[4] J. Aycock, A brief history of just-in-time, ACM Comput. Surv. 35 (2) (2003) 97–113. `doi:10.1145/857076.857077`.

[5] ECMA International, Standard ECMA-262 - ECMAScript Language Specification, 5th Edition, 2011.

[6] C. Chambers, D. Ungar, E. Lee, An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes, Lisp and Symbolic Computation 4 (3) (1991) 243–281.

[7] J. R. Bell, Threaded code, Commun. ACM 16 (6) (1973) 370–372. `doi:10.1145/362248.362270`.

[8] M. A. Ertl, D. Gregg, A. Krall, B. Paysan, Vmgen - a generator of efficient virtual machine interpreters, Softw., Pract. Exper. 32 (3) (2002) 265–294. `doi:10.1002/spe.434`.
URL `https://doi.org/10.1002/spe.434`

[9] D. Gregg, M. A. Ertl, A language and tool for generating efficient virtual machine interpreters, in: Proc. Domain-Specific Program Generation, 2003, pp. 196–215. doi:10.1007/978-3-540-25935-0_12.
URL https://doi.org/10.1007/978-3-540-25935-0_12

[10] M. A. Ertl, A portable forth engine, in: Proc. EuroFORTH '93 Conference, 1993.

[11] D. Gregg, M. A. Ertl, A. Krall, Implementing an efficient Java interpreter, in: Proc. High-Performance Computing and Networking, 9th International Conference, HPCN Europe 2001, 2001, pp. 613–620. doi:10.1007/3-540-48228-8_70.
URL https://doi.org/10.1007/3-540-48228-8_70

[12] A. Beatty, K. Casey, D. Gregg, A. Nisbet, An optimized Java interpreter for connected devices and embedded systems, in: Proc. 2003 ACM Symposium on Applied Computing, SAC 2003, ACM, 2003, pp. 692–697. doi:10.1145/952532.952667.

[13] K. Tanaka, A. D. Nagumanthri, Y. Matsumoto, mruby – rapid software development for embedded systems, in: Proc. 15th International Conference on Computational Science and Its Applications, ICCSA 2015, 2015, pp. 27–32. doi:10.1109/ICCSA.2015.22.
URL https://doi.org/10.1109/ICCSA.2015.22

[14] T. Azumi, Y. Nagahara, H. Oyama, N. Nishio, mruby on TECS: component-based framework for running script program, in: Proc. IEEE 18th International Symposium on Real-Time Distributed Computing, ISORC 2015, 2015, pp. 252–259. doi:10.1109/ISORC.2015.42.

[15] S. Brunthaler, Inline caching meets quickening, in: Proc. Object-Oriented Programming, 24th European Conference, ECOOP 2010, 2010, pp. 429–451. doi:10.1007/978-3-642-14107-2_21.

[16] S. Brunthaler, Efficient interpretation using quickening, in: Proc. 6th Symposium on Dynamic Languages, DLS 2010, ACM, 2010, pp. 1–14. doi:10.1145/1869631.1869633.

[17] T. A. Proebsting, Optimizing an ANSI C interpreter with superoperators, in: Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, 1995, pp. 322–332. doi:10.1145/199448.199526.
URL http://doi.acm.org/10.1145/199448.199526

[18] G. B. Prokopski, C. Verbrugge, Analyzing the performance of code-copying virtual machines, in: Proc. 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, ACM, 2008, pp. 403–422. doi:10.1145/1449764.1449796.
URL http://doi.acm.org/10.1145/1449764.1449796

[19] B. Klemens, 21st Century C: C Tips from the New School, 1st Edition, O'Reilly Media, Inc., 2012.

[20] C. Kim, J. Kim, S. Kim, D. Kim, N. Kim, G. Na, Y. H. Oh, H. G. Cho, J. W. Lee, Typed architectures: Architectural support for lightweight scripting, in: Proc. 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, ACM, New York, NY, USA, 2017, pp. 77–90. `doi:10.1145/3037697.3037726`.
URL `http://doi.acm.org/10.1145/3037697.3037726`

[21] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, C. Wimmer, Self-optimizing AST interpreters, in: Proc. 8th Symposium on Dynamic Languages, DLS 2012, ACM, 2012, pp. 73–82. `doi:10.1145/2384577.2384587`.
URL `http://doi.acm.org/10.1145/2384577.2384587`

[22] C. Humer, C. Wimmer, C. Wirth, A. Wöß, T. Würthinger, A domain-specific language for building self-optimizing AST interpreters, in: Proc. 2014 International Conference on Generative Programming: Concepts and Experiences, GPCE 2014, 2014, pp. 123–132. `doi:10.1145/2658761.2658776`.
URL `http://doi.acm.org/10.1145/2658761.2658776`

[23] N. D. Jones, Transformation by interpreter specialisation, Sci. Comput. Program. 52 (2004) 307–339. `doi:10.1016/j.scico.2004.03.010`.
URL `https://doi.org/10.1016/j.scico.2004.03.010`

[24] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, G. Muller, Static and dynamic program compilation by interpreter specializ ation, Higher-Order and Symbolic Computation 13 (3) (2000) 161–178. `doi:10.1023/A:1010078412711`.
URL `https://doi.org/10.1023/A:1010078412711`

[25] D. E. Knuth, The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams, 12th Edition, Addison-Wesley Professional, 2009.

[26] E. Dujardin, E. Amiel, E. Simon, Fast algorithms for compressed multi-method dispatch table generation, ACM Trans. Program. Lang. Syst. 20 (1) (1998) 116–165. `doi:10.1145/271510.271521`.
URL `http://doi.acm.org/10.1145/271510.271521`