SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2187

From bytecode to JavaScript: the Js_of_ocaml compiler

Jérôme Vouillon[1,*,†] and Vincent Balat[2]

[1]*CNRS, PPS, UMR 7126, Université Paris Diderot, Sorbonne Paris Cité, F-75205 Paris, France*
[2]*Université Paris Diderot, Sorbonne Paris Cité, PPS UMR 7126, CNRS, INRIA Paris-Rocquencourt, F-75205 Paris, France*

SUMMARY

We present the design and implementation of a compiler from OCaml bytecode to JavaScript. The compiler first translates the bytecode into a static single-assignment intermediate representation on which optimizations are performed, before generating JavaScript. We believe that taking bytecode as an input instead of a high-level language is a sensible choice. Virtual machines provide a very stable API. Such a compiler is thus easy to maintain. It is also convenient to use, and it can just be added to an existing installation of the development tools. Already-compiled libraries can be used directly, with no need to reinstall anything. Finally, some virtual machines are the target of several languages. A bytecode to JavaScript compiler would make it possible to retarget all these languages to Web browsers at once. Copyright © 2013 John Wiley & Sons, Ltd.

Received 5 April 2012; Revised 17 August 2012; Accepted 15 January 2013

KEY WORDS: compiler; OCaml; JavaScript; bytecode

1. INTRODUCTION

We present a compiler translating OCaml [1] bytecode into JavaScript [2]. This compiler makes it possible to program client-side interactive Web applications in OCaml.

JavaScript is the only language that is readily available in most Web browsers and that provides a direct access to browser APIs. (Other platforms, such as Flash and Silverlight, are not as widely available nor as integrated.) It is thus the mandatory language for developing Web applications. Still, it would be interesting to be able to use a variety of languages on a Web browser: JavaScript may be suitable for some tasks, but other languages can be more appropriate in other cases. In particular, being able to use the same language both on browsers and servers makes it possible to share code and to reduce the language impedance mismatch between the two tiers. For instance, form validation must be performed on the server for security reasons and is desirable on the client to provide an early feedback to the user. Having to maintain two different pieces of code performing essentially the same task in two different languages is error-prone. When using a single language, the impedance mismatch in client–server communication is also greatly reduced: data still has to be marshaled, but the same type definitions can be used on both side, with no need for translation. In our case, we have appreciated being able to share a large amount of code when developing a graph viewer application with both a GTK and a Web user interface.

Additionally, thanks to recent works on highly optimized interpreters using just-in-time compilation, JavaScript now exhibits decent performance. For all these reasons, it is a sensible target for a compiler.

*Correspondence to: Jérôme Vouillon, Laboratoire PPS, Université Paris Diderot, Case 7014, 75205 Paris Cedex 13, France.
†E-mail: jerome.vouillon@gmail.com

We have chosen to take OCaml bytecode as an input, rather than a source code, based on maintenance and ease-of-use considerations. Indeed, we have very limited human resources available, and we are targeting a small community of developers. Virtual machines provide a very stable API. The JVM [3]) and .NET Common Language Runtime hardly change, whereas the source languages continue to evolve. The same is true for the OCaml virtual machine. Hence, there is no need to modify the compiler at each release of the language to support the latest features (or just for it to continue to work, if it were implemented as patches against the main compiler). This is crucial for us. We have seen too many interesting OCaml-related projects die because of lack of maintenance: OCamlIl [4] (a compiler to .NET), OCamlExc [5] (a static analyzer of spurious exceptions), ocamldefun (a defunctorizer), and so on. In addition, with our compiler, the barrier to entry is low for programmers. A programmer wanting to target Web browsers can just install the compiler as an add-on to its usual OCaml development environment, rather than installing a specific development environment for the Web. In particular, already-installed precompiled libraries can be used directly. Finally, whereas this is not the case for the OCaml virtual machine, some virtual machines are the target of many languages. For the JVM, one can list, among many others, Java, Scala, Clojure, and JRuby. A single compiler from bytecode to JavaScript would provide a tight integration of all these languages in Web browsers. This is in contrast, for instance, with the Google Web Toolkit [6] that makes it possible to run Java programs but not Scala programs on a browser and misses the latest Java features.

There are challenges to address when starting from bytecode rather than source code. First, the data representation is low level. For instance, functions have been compiled down to flat closures; the bytecode interpreter is a stack machine. Also, little type information remains to help us in the translation. It was not clear at first whether these data representations could be mapped to available JavaScript data structures in an efficient way. Second, one may fear that going from a low-level language to a higher-level language would result in a low code density. Third, one must find ways to represent unstructured code using the limited JavaScript control statements (JavaScript does not have a `goto` statement). Last, one must design a way to use the available JavaScript APIs in an easy way, although they are object-oriented and the calling convention of JavaScript differs from the OCaml one. We believe that we have addressed these challenges successfully and that starting from OCaml bytecode provides a good trade-off.

One of the design goals for the compiler was to rapidly have a working implementation that yet provides a solid basis for future developments. Thus, at the moment, no sophisticated optimization has been implemented. The focus has rather been on simple but effective analyses and code transformations, designed to achieve good performance, and also to generate compact code. Indeed, the compiled programs are intended to be transferred a large number of times over the network. It is thus important to minimize latency (Web page loading times) and bandwidth usage.

The compilation process is fairly standard. We first present the OCaml data structures and how they are represented in JavaScript (Section 2). Bytecode programs are converted to a static single-assignment (SSA)-based intermediate form (Section 3). Some optimizations are performed on the intermediate code (Section 4). Then, the intermediate code is translated to JavaScript (Section 5). We document some OCaml-specific issues in Section 6. The compiler would not be usable without ways of manipulating JavaScript values and accessing browser APIs. We deal with this interoperability issue in Section 7. We have performed extensive benchmarks of the compiler to assess its performance (Section 8). Finally, Sections 9 and 10 present related and future works.

2. DATA REPRESENTATION

Data representation is crucial performance-wise. One must choose representations that match the OCaml semantics and that are implemented efficiently by JavaScript engines.

OCaml has a number of predefined types: integers, floating-point numbers, characters, strings, booleans, and arrays. New types can be declared using a `type` declaration. For instance, the following declaration defines trees of integers.

```
type tree = Leaf | Node of tree * int * tree
```


DOI: 10.1002/spe

A value of type `tree` is either a leaf, with constructor `Leaf`, or a node, with constructor `Node`, containing two subtrees and an integer.

The OCaml virtual machine differs only slightly from the Zinc machine [7] of Caml Light. It uses a very uniform memory model. An OCaml value is a word of either 32 or 64 bits depending on the architecture. This word represents either an integer or a pointer in the heap (integers are the only unboxed values). They are distinguished by the lower bit, which is 1 for integers and 0 for pointers. The heap is composed of memory blocks of arbitrary size preceded by a one-word header. This header contains pieces of information such as the size of the block, a tag indicating the kind of the block, and some bits reserved for the garbage collector. One of the uses of the tag is to distinguish structured blocks (containing valid values), which should be recursively traversed by the garbage collectors from blocks containing unstructured data (such as floating-point numbers or the characters of a string).

With the OCaml virtual machine, integers, booleans, and characters are mapped to integers. Arrays are mapped to structured blocks. String and floating-point numbers are stored into unstructured blocks. Functional values are mapped to flat closures, that is, blocks with a special tag and that contain a pointer to the code of the function and the values of the free variables of the function. Constructors with no argument, such as `Leaf`, are mapped to integers. Other type constructors are mapped to memory blocks. The integer value and the tag of the memory block make it possible to distinguish the different constructors of a same type definition. Modules are mapped to memory blocks; functors (i.e., higher-order modules) are mapped to functions taking modules as arguments and returning a module.

For the translation to JavaScript, we have made the following choices. The integers and floating-point numbers of the OCaml virtual machine are mapped to JavaScript numbers. Structured blocks are mapped to JavaScript arrays. The first element of these arrays is the tag; subsequent elements are the contents of the block. Closures are mapped to JavaScript functions. We take advantage of the scoping mechanism of JavaScript. The function body is compiled in such a way that free variables are accessed directly from the outer scopes rather than from the closure. We use our own implementation of strings. Indeed, OCaml strings are mutable arrays of eight-bit characters, whereas JavaScript strings are immutable UTF-16 strings. More details on how integers and strings are handled are given in Section 6.

We do not perform any special mapping for exceptions. OCaml exceptions are thus not instances of JavaScript exception objects. Their semantics is faithfully implemented; in particular, they remain generative. OCaml objects are mostly compiled away during bytecode generation. Thus, there is not much to do to support them. There are just a few bytecode instructions for method resolution, which are implemented as JavaScript functions.

3. FROM BYTECODE TO INTERMEDIATE CODE

3.1. OCaml bytecode

The OCaml virtual machine [7] is a stack machine (like the JVM) with an accumulator (a single register that stores the result of the last instruction, if any, thus avoiding some stack operations).

A bytecode program is basically composed of a sequence of instructions ending by a `STOP` instruction. We list in Figure 1 some of the bytecode instructions. We use them to illustrate the compilation process. The semantics of these operations is given in Section 3.3 when presenting the conversion from bytecode to intermediate code. In the actual bytecode, the instruction `BGEINT` only takes one target address `k`. The second address `k'` is convenient for specifying the translation to intermediate code. It will always be the address of the immediately following instruction.

As a running example, we consider the OCaml code sample. A function `f` is defined. This function takes as argument an integer `x`. If the integer is strictly positive, the function returns twice the integer. Otherwise, it returns the integer itself. The function is later applied to integer `10`.

```
let f(x) = if x > 0 then 2 * x else x f(10)
```


DOI: 10.1002/spe

bytecode stream	B	::=	I ; B | `STOP`	
bytecode instruction	I	::=	`ACC0` | `ACC1` | `PUSH`	*stack manipulation*
		|	`CONSTINT n` | `MULINT`	*integer operations*
		|	`CLOSURE n,k` | `APPLY1`	*function operations*
		|	`RETURN n`	
		|	`BRANCH k` | `BGEINT n,k,k'`	*branch instructions*
		|	...	

Figure 1. Bytecode instructions.

```
68 ACC0              copy top of stack to accu
69 BGEINT 0,79       branch if 0 ≥ accu
72 ACC0
73 PUSH              push accu on stack
74 CONSTINT 2        store integer 2 into accu
76 MULINT            multiply the two values
77 RETURN 1          pop stack and return
79 ACC0
80 RETURN 1

82 CLOSURE 0,68      allocate closure
85 PUSH
86 CONSTINT 10
88 PUSH
89 ACC1              copy second element of stack to accu
90 APPLY1            invoke function
   ...
```

Figure 2. Bytecode sample.

The decompiled portion of a bytecode program corresponding to these two lines is shown in Figure 2. The leftmost column is the address (in words) of each instruction. The bytecode is generated out of order. The function bodies are produced first, before the code corresponding to the outer context. Hence, the execution of this piece of code does not start at address 68, which is the start address of the function body. Rather, when running the program, the execution moves at some point to address 82. There, the function closure corresponding to function `f` is allocated and put in the accumulator. The function has no free variable; hence, its environment is empty (integer argument 0). The code of the function starts at address 68. The closure is pushed on the stack (instruction `PUSH`). Then, the function call is performed. The integer constant 10 is loaded in the accumulator, then pushed on the stack. The closure is retrieved from the stack and put in the accumulator (instruction `ACC1`). Finally, the function is invoked (instruction `APPLY1`). The execution process thus moves to address 68. The integer 10 at the top of the stack is copied into the accumulator. As it is not inferior or equal to 0, the conditional branch is not taken. The integer is put in the accumulator, then pushed again onto the stack. The integer 2 is placed into the accumulator. The instruction `MULINT` multiplies the two integers in the accumulator and on the stack top and pops the stack. The resulting integer 20 is in the accumulator. Finally, the function returns, discarding one value (integer 10) from the stack. The value 20 returned by the function is in the accumulator.

3.2. Intermediate code

A variant of SSA form is used as an intermediate representation. This variant has a more functional flavor than the standard SSA presentation. When two control-flow edges join, a variable may hold different values depending on the incoming edge. In standard SSA form, this is expressed by a notational trick, the ϕ function: $x = \phi(y, z)$ means that the value of either variable y or variable z is assigned to variable x, depending on the incoming edge. Here, instead of using ϕ functions, blocks are parameterized, and values are passed explicitly from blocks to blocks. For instance, if there is a jump from a source block with arguments z and t to a target block with two parameters x and y, the values of variables z and t are assigned, respectively, to variables x and y when moving from

DOI: 10.1002/spe

intermediate code	C	::=	$i\,;C \mid c$	
instruction	i	::=	$x = e$	*assignment*
		|	$\ldots$	
expression	e	::=	n	*integer constant*
		|	$\mathrm{fun}(\sigma)\{\kappa(\sigma')\}$	*function closure*
		|	$x(\sigma)$	*function invocation*
		|	"p"(σ)	*primitive invocation*
		|	$\ldots$	
control instruction	c	::=	branch $\kappa(\sigma)$	*unconditional branch*
		|	if x then $\kappa(\sigma)$ else $\kappa'(\sigma')$	*conditional branch*
		|	return x	*function return*
		|	stop	*end of program*
		|	$\ldots$	
block parameters	σ	::=	$x_1, \ldots, x_n$	

Figure 3. Intermediate code.

the source block to the destination block. As we shall see at the beginning of Section 4, there is a simple correspondence between the two notations: a ϕ function can be associated to each block parameter. Our variant otherwise keeps the essential property that each variable in a program has only one definition (it is assigned to only once). The domination-based scoping of SSA is also kept: the scope of a variable extends from its definition to every places that can be reached only via its definition.

The intermediate representation is composed of a set of blocks. A block is composed of a block location κ, the block parameters σ, and some code C. The notation $\kappa(\sigma)$ is used for a jump to location κ with arguments σ. The compiler uses integers for block locations (this integer is in fact the location of the corresponding bytecode sequence in the source program). Block parameters σ are a sequence of variables $x_1, \ldots, x_n$. A variable x is represented by an integer. A counter is used in the implementation to generate fresh variables.

The syntax of intermediate code is given in Figure 3. A piece of intermediate code C is a sequence of instructions i followed by a control instruction c. One instruction is the assignment of the value of an expression e to a variable x. Expressions can be, among others, a constant, a function closure, a function invocation, or a primitive invocation. Integer multiplication is one of these primitives. Later, we present other instructions to manage exceptions (Section 3.4) and deal with memory blocks (Section 4.3).

3.3. Translation to intermediate code

The bytecode program is split by the compiler in blocks of bytecode instructions that are always executed sequentially. Each block is compiled to intermediate code independently. The first step of the compilation process is to delimit these blocks. The start address of each block can be found by traversing linearly the code: it is either the address 0 (program entry point), the target of a branch instruction, the start of a function, or the address of an exception handler (see Section 3.4 for the latter). The end of a block can be explicitly indicated by a control instruction. It may also be implicit, when a branch instruction points to the middle of a sequence of instructions. All these implicit delimitations are collected by scanning the bytecode sequentially from start to end and recording the target address of each branch instruction.

To compile a block, one also needs its static environment. More precisely, one needs the size of the current stack frame. For this, a recursive traversal of the program is performed, starting from the entry point, following the branches, and visiting the function body of each closure, while keeping track of the size of the current stack frame.

We can now specify the translation of a block. The contents of the accumulator are represented symbolically by a variable x, whereas the contents of the current stack frame are represented by a sequence of variables σ. Values flow from previous blocks through the accumulator and the stack. Hence, the parameters of a block are a concatenation x, σ where x and σ are composed

DOI: 10.1002/spe

$$\frac{x\,;\,x,\sigma \vdash B \rightsquigarrow C}{y\,;\,x,\sigma \vdash \texttt{ACC0}\,;\,B \rightsquigarrow C} \qquad \frac{y\,;\,x,y,\sigma \vdash B \rightsquigarrow C}{z\,;\,x,y,\sigma \vdash \texttt{ACC1}\,;\,B \rightsquigarrow C} \qquad \frac{x\,;\,x,\sigma \vdash B \rightsquigarrow C}{x\,;\,\sigma \vdash \texttt{PUSH}\,;\,B \rightsquigarrow C}$$

$$\frac{x\,;\,\sigma \vdash B \rightsquigarrow C \qquad x \text{ fresh}}{y\,;\,\sigma \vdash \texttt{CONSTINT n}\,;\,B \rightsquigarrow x = \texttt{n}\,;\,C} \qquad \frac{z\,;\,\sigma \vdash B \rightsquigarrow C \qquad z \text{ fresh}}{x\,;\,y,\sigma \vdash \texttt{MULINT}\,;\,B \rightsquigarrow z = \text{“}*\text{”}(x,y)\,;\,C}$$

$$x\,;\,\sigma \vdash \texttt{BRANCH k}\,;\,B \rightsquigarrow \text{branch } \texttt{l}(x,\sigma)$$

$$\frac{y \text{ fresh}}{x\,;\,\sigma \vdash \texttt{BGEINT n,k,k'}\,;\,B \rightsquigarrow y = \texttt{n} \geq x\,;\,\text{if } y \text{ then } \texttt{l}(x,\sigma) \text{ else } \texttt{l}'(x,\sigma)}$$

$$\frac{z\,;\,\sigma \vdash B \rightsquigarrow C \qquad z \text{ fresh}}{x\,;\,y,\sigma \vdash \texttt{APPLY1}\,;\,B \rightsquigarrow z = x(y)\,;\,C}$$

$$\frac{y\,;\,\sigma \vdash B \rightsquigarrow C \qquad y \text{ and } \sigma' \text{ fresh} \qquad |\sigma'| = \text{arity}(\texttt{l})}{x\,;\,\sigma \vdash \texttt{CLOSURE 0,k}\,;\,B \rightsquigarrow y = \text{fun}(\sigma')\{\texttt{k}(\diamond,\sigma')\}\,;\,C} \qquad x\,;\,\sigma \vdash \texttt{RETURN n}\,;\,B \rightsquigarrow \text{return } x$$

$$x\,;\,\sigma \vdash \texttt{STOP} \rightsquigarrow \text{stop}$$

Figure 4. Translation to intermediate code.

of fresh variables. The translation of a block, specified in Figure 4 using inductive rules, then takes this pair as an input. It is defined as a predicate $x;\sigma \vdash B \rightsquigarrow C$ where B is the sequence of bytecode instructions to compile, and C is the resulting intermediate code. The idea of starting the translation of each block with fresh variables can also be found in [8] and [9], which propose algorithms to put code into SSA form. For specifying the translation, we assume that whenever a block is implicitly terminated in the actual bytecode, a `BRANCH` instruction has been added to its tail, pointing to the immediately following block. The actual implementation compares the current location in the bytecode to the end of the block and generates a branch instruction when the limit is reached.

We present the translation of the most interesting instructions. No code is produced when translating a stack access `ACC0`. One just records that afterward the value of the accumulator is the top element of the stack. The `BRANCH k` instruction is compiled to a branch instruction. The target block of the branch is at location `k`. The block arguments are the concatenation of the contents of the accumulator and of the stack. The `APPLY1` instruction is compiled to the call of the function contained in the accumulator applied to a single parameter at the top of the stack. The results of the call is stored in a fresh variable z. Afterward, the top element of the stack is discarded, and the accumulator contains the value returned by the function. The `CLOSURE` instruction is compiled to a function closure allocation. We show here only the translation when the function has no free variable (when its environment is empty). The closure parameters are a sequence of fresh variables σ'. Their number is given by the arity of the function, which can be obtained from the location of its body `k`. The arguments are expected on the stack. The contents of the accumulator are unknown at the beginning of a function body and will not be used. Hence, a dummy variable $\diamond$ is used to symbolize this contents. The closure is stored in a fresh variable y. Afterward, the accumulator contains the closure. The `RETURN` instruction is compiled to a corresponding return instruction. The number `n` of stack items to be discarded is not useful after translation. The returned value x is in the accumulator.

When compiling a function, one actually needs to deal with the environment of the current function, which contains the values of the free variables of the function. We only sketch the translation of functions with nonempty environments. A bytecode closure is a memory block containing a pointer to the function code followed by the value of each free variable of the function. The `CLOSURE n,k` instruction grabs the `n` topmost elements of the stack to build the closure. When executing the body of a function, its closure is stored in a specific environment register. The `ENVACC n` instruction copies the `n`th element of the environment to the accumulator. The compiler takes advantage of JavaScript static scoping. For this, the environment is eliminated in a way similar to the stack. The translation predicate $x;\sigma;\eta \vdash B \rightsquigarrow C$ takes an additional argument η representing symbolically the contents of the environment register. This argument is a sequence of variables each standing for an element of the environment. The translation of the `ENVACC` instruction is then

DOI: 10.1002/spe

Accu and stack	Bytecode	Intermediate code
		block $68(a, b)$
$a\,;b$	`68 ACC0`	
$b\,;b$	`69 BGEINT 0,79`	$c = 0 \geq b$
		if c then $79(b, b)$
		else $72(b, b)$
		block $72(d, e)$
$d\,;e$	`72 ACC0`	
$e\,;e$	`73 PUSH`	
$e\,;e, e$	`74 CONSTINT 2`	$f = 2$
$f\,;e, e$	`76 MULINT`	$g =$ "$*$"(f, e)
$g\,;e$	`77 RETURN 1`	return g
		block $79(h, i)$
$h\,;i$	`79 ACC0`	
$i\,;i$	`80 RETURN 1`	return i
		block $82(j)$
$j\,;\bullet$	`82 CLOSURE 0,68`	$l = \mathrm{fun}(m)\{68(\diamond, m)\}$
$l\,;\bullet$	`85 PUSH`	
$l\,;l$	`86 CONSTINT 10`	$n = 10$
$n\,;l$	`88 PUSH`	
$n\,;n, l$	`89 ACC1`	
$l\,;n, l$	`90 APPLY1`	$o = l(n)$
$o\,;l$	...	

Figure 5. Example of translation.

similar to the `ACCn` instructions, except that the new contents of the accumulator are taken from the environment rather than from the stack.

$$\frac{y; \sigma; \eta \vdash B \rightsquigarrow C \quad \eta(\mathtt{n}) = y}{x; \sigma; \eta \vdash \mathtt{ENVACC}\ \ \mathtt{n}; B \rightsquigarrow C}$$

As an example, the translation of the piece of bytecode in Figure 2 is given in Figure 5. The translation process starts at address 82, which corresponds to the closure allocation and the function application. (There is a branch instruction pointing to this address, not shown here.) We assume that the stack is empty at the beginning of the block (we write $\bullet$ for an empty sequence of variables).

3.4. Exceptions

Three bytecode instructions are dedicated to exception handling.

$$I ::= \ldots \mid \mathtt{RAISE} \mid \mathtt{PUSHTRAP\ k} \mid \mathtt{POPTRAP}$$

The virtual machine keeps in a register a pointer to the stack frame describing the current exception handler, if any. This stack frame contains the address of the handler, the current function environment, and a pointer to the frame of the previous handler. The `PUSHTRAP` instruction installs a handler by recording such a frame. The `POPTRAP` instruction restores the previous handler and pops the frame. The `RAISE` instruction pops all elements of the stack including the stack frame of the current handler. It restores the previous handler. Finally, it jumps to the handler. It expects the raised exception to be in the accumulator. The intermediate code has corresponding control instructions.

$$\begin{aligned} c ::= {} & \mathrm{pushtrap}\ (\kappa_1(\sigma_1), \lambda x.\, \kappa_2(\sigma_2)) \\ & \mid \mathrm{poptrap}\ \kappa(\sigma) \mid \mathrm{raise}\ x \mid \ldots \end{aligned}$$

Besides, an exception handler $(x, (\kappa_2, \sigma))$ is associated to each block in between a `PUSHTRAP` and the matching `POPTRAP`. The arguments σ to the exception handler are the bound variable x standing for the raised exception and the variables corresponding to the portion of the stack available to the handler. These arguments change from block to block, as fresh variables are used each time. This associated handler makes it explicit which variables of a block are passed to the handler, which is crucial for correct code analyses.

DOI: 10.1002/spe

4. CODE ANALYSES AND TRANSFORMATIONS

Several code analyses and transformations are performed on the intermediate code to improve the performance of the generated code and reduce its size. The main issue, performance-wise, is the calling convention mismatch between OCaml, which encourages a curried style, and JavaScript, which does not support currying. To deal with this, the closures possibly involved at each call point are computed (Section 4.3). Then, optimized function calls are generated when the closure arities match the number of arguments provided (see Section 6 for details). Self-tail calls are very common and should be implemented using constant stack depth (Section 4.1). Finally, dead code elimination is performed (Section 4.4). A difficulty is that all the components of a module are stored in a common memory block. Thus, if there is any reference to this block, a rough dead code elimination algorithm would retain the contents of the whole module. We thus use an algorithm that is aware of structured memory blocks. The strategy used is to eliminate references to these blocks by replacing field accesses by direct reference to the field contents (Section 4.3).

At the moment, the compiler only uses intraprocedural analyses. Indeed, they are much simpler to implement and are already quite effective. To define some of the code analyses, it is convenient to associate to each variable x of the program its 'definition' $\mathrm{def}(x)$ as follows:

- $\mathrm{def}(x) = e$ if there exists an instruction $x = e$ in the program;
- $\mathrm{def}(x) = \phi(x_1, \ldots, x_n)$ if variable x is a block parameter and variables x_1 to x_n are the possible corresponding arguments (this is the ϕ function of standard SSA form);
- $\mathrm{def}(x) = \star$ if x is a function parameter.

As each variable is assigned only once, this defines a total function over variables. We consider the arguments of the ϕ function as a set of variables. Thus, for instance, $\phi(x, x) = \phi(x)$.

4.1. Self-tail call optimization

Tail recursion optimization is performed first. Whenever a function f calls itself in tail position, the call is replaced by a branch to the beginning of the function. Formally, suppose we have a function definition

$$f = \mathrm{fun}(\sigma_1)\{\kappa(\sigma_2)\}$$

If one of the blocks of the function body ends with

$$x = f(\sigma_3); \quad \text{return } x$$

then, these two instructions are replaced by the following:

$$\text{branch } \kappa(\sigma_4)$$

where the arguments σ_4 are built from the block arguments σ_2 by replacing any variable in the function parameters σ_1 by the corresponding variable in the function arguments σ_3.

4.2. Minimizing variable passing between blocks

So far, the number of parameters of a block is equal to the depth of the stack frame at this corresponding point in the bytecode program. By a suitable renaming of variables, the number of arguments passed from one block to its successors can be greatly reduced. The compiler follows the approach in [9]. Whenever one has $\mathrm{def}(x) = \phi(y)$ or $\mathrm{def}(x) = \phi(x, y)$, one can replace all usages of variable x by variable y. Indeed, any value assigned to variable x must in both cases have been assigned to variable y beforehand. Then, the block parameter x can be eliminated. By performing this operation repeatedly until no such definition remains, one reduces the number of ϕ-functions (i.e., the number of block parameters) in the program. It can be proved that this algorithm is correct and besides that it computes the minimal ϕ-function placement for reducible control-flow graphs [9], which the OCaml compiler always generates. The implementation does not actually perform variable substitution eagerly. A union-find data structure is used to keep track

DOI: 10.1002/spe

```
block 68()
  c = 0 ≥ m
  if c then 79() else 72()

block 72()
  f = 2
  g = "*"(f, m)
  return g

block 79()
  return m

block 82()
  l = fun(m){68()}
  n = 10
  o = k(n)
  ...
```

Figure 6. Code after redundant variable removal.

of which variable should be replaced by which. A global substitution is performed once no more simplification is possible. The removal of unused block parameters is also not performed immediately, but by dead code elimination (Section 4.4). Figure 6 shows the result of the transformation (together with dead code elimination) applied to the code in Figure 5. In this example, all block parameters are eliminated.

4.3. Data flow analysis

The compiler performs an analysis to determine an overapproximation of which values may be contained in each variable. This analysis has to deal in a sound fashion with mutable fields and function calls, including calls to arbitrary external functions. Thus, it combines a flow-insensitive data flow analysis with an escape analysis that computes which values might be modified. A major use of the analysis is for shortcutting memory block accesses, replacing field accesses by direct references to the contents of the field. If the compiler is able to remove all accesses to a given memory block, then the block does not have to be allocated anymore. Besides, then, any value stored in one of its fields but not otherwise used also becomes unnecessary. As OCaml modules are implemented as memory blocks, this optimization is crucial for effective dead code elimination. The analysis is also used to generate optimized code for some operations: function calls (Section 6), integer multiplication (Section 6), JavaScript method invocations (Section 7), and others.

The analysis deals with the following memory block operations in a special way . The expression $[i\,|x_1, \ldots, x_n]$ of the intermediate code allocates a memory block with a tag i and n fields whose values are given by variables x_1 to x_n. The expression $x[i]$ accesses field i of the memory block contained in value x. The instruction $x[i] = y$ stores the value of variable y in field i of the memory block contained in value x.

$$
\begin{array}{lll}
e ::= & \ldots & \\
 & \mid\ [i\,|x_1, \ldots, x_n] & \text{block allocation} \\
 & \mid\ x[i] & \text{field access} \\
i ::= & \ldots & \\
 & \mid\ x[i] = y & \text{field update}
\end{array}
$$

The analysis consists in computing two predicates. Predicate $x \leftarrow y$ indicates that variable x may contain values coming from variable y. Predicate $x \leftarrow ?$ holds when x may contain values of unknown origin. Thus, when $x \leftarrow ?$ does not hold, the only possible values for variable x are the values of expressions $\text{def}(y)$ where $x \leftarrow y$.

The first step of the analysis is the computation of predicate $x \leftarrow y$, as specified in Figure 7. The idea is to track how values flow through code blocks and memory blocks. Basically, source variables are propagated through block parameters (first rule) and field accesses (second rule). In all other cases, we take $x \leftarrow x$. The implementation collects for each variable x the set of variables y such that $x \leftarrow y$. A standard work list algorithm is used. A pitfall is that the dependency graph between variables is not fully known initially and has to be updated dynamically. Indeed, for the second rule, whenever one learns that $y \leftarrow z$, one must add a dependency of variable x on the corresponding variable x_i.

As a second step, an escape analysis is performed. This is specified as predicate 'x escapes' in Figure 8. In this figure, we write $i \in \mathcal{P}$ to mean that the instruction i occurs somewhere in

DOI: 10.1002/spe

$$\frac{\mathrm{def}(x) = \phi(x_1, \ldots, x_n) \qquad x_i \hookleftarrow y}{x \hookleftarrow y} \qquad \frac{\begin{array}{c}\mathrm{def}(x) = y[i] \qquad y \hookleftarrow z \\ \mathrm{def}(z) = [j|x_1, \ldots, x_i, \ldots, x_n] \end{array} \qquad x_i \hookleftarrow t}{x \hookleftarrow t}$$

$$\frac{\begin{array}{c}\mathrm{def}(x) = e \\ e \text{ not of the shape } y[i]\end{array}}{x \hookleftarrow x} \qquad \frac{\mathrm{def}(x) = \star}{x \hookleftarrow x}$$

Figure 7. Propagation of known values.

$$\frac{\begin{array}{c}\mathrm{def}(x) = y(\ldots, z, \ldots) \\ z \hookleftarrow t\end{array}}{t \text{ escapes}} \qquad \frac{\begin{array}{c}\mathrm{def}(x) = \text{“}p\text{”}(\ldots, z, \ldots) \\ z \hookleftarrow t\end{array}}{t \text{ escapes}} \qquad \frac{\text{return } z \in \mathcal{P} \qquad z \hookleftarrow t}{t \text{ escapes}}$$

$$\frac{\text{raise } z \in \mathcal{P} \qquad z \hookleftarrow t}{t \text{ escapes}} \qquad \frac{\begin{array}{c}x[i] = z \in \mathcal{P} \\ z \hookleftarrow t\end{array}}{t \text{ escapes}} \qquad \frac{x \text{ escapes} \qquad \begin{array}{c}\mathrm{def}(x) = [j|x_1, \ldots, x_n] \\ x_i \hookleftarrow y\end{array}}{y \text{ escapes}}$$

$$\frac{x \text{ escapes}}{x \text{ mutable}} \qquad \frac{x[i] = y \in \mathcal{P} \qquad x \hookleftarrow z}{z \text{ mutable}}$$

Figure 8. Escaping values.

$$\frac{\begin{array}{c}\mathrm{def}(x) = \phi(x_1, \ldots, x_n) \\ x_i \hookleftarrow ?\end{array}}{x \hookleftarrow ?} \qquad \frac{\begin{array}{c}\mathrm{def}(x) = y[i] \\ y \hookleftarrow ?\end{array}}{x \hookleftarrow ?} \qquad \frac{\begin{array}{c}\mathrm{def}(x) = y[i] \qquad y \hookleftarrow z \\ \mathrm{def}(z) \text{ not of the shape } [j|x_1, \ldots, x_i, \ldots, x_n]\end{array}}{x \hookleftarrow ?}$$

$$\frac{\mathrm{def}(x) = y[i] \qquad y \hookleftarrow z \qquad z \text{ mutable}}{x \hookleftarrow ?} \qquad \frac{\begin{array}{c}\mathrm{def}(x) = y[i] \qquad y \hookleftarrow z \\ \mathrm{def}(z) = [j|x_1, \ldots, x_i, \ldots, x_n] \qquad x_i \hookleftarrow ?\end{array}}{x \hookleftarrow ?}$$

Figure 9. Propagation of unknown values.

the whole program. The goal is to determine which memory blocks may be modified, which is predicate ‘x mutable’ in the same figure. If a variable z either occurs as a parameter of a function or a primitive, is returned, is raised, or is assigned to the field of a block, then all its possible known values, given by variables t such that $z \hookleftarrow t$, escape (first five rules). If a block escapes, then the values of all its fields also escape (sixth rule). A value is considered mutable if either it escapes or it is the target of a block update (last two rules). The two predicates can be computed using recursive functions.

Finally, the predicate $x \hookleftarrow ?$ specified in Figure 9 can be computed. It indicates which variable x may contain other unknown values besides the values given by predicate $x \hookleftarrow y$. If variable x_i may contain unknown values and is assigned to variable x (by branching to a code block), then variable x may contain unknown values (first rule). If we access a block y but do not know enough information regarding the accessed field, either because not all possible shapes for block y are known precisely or the field contents may have been modified, then we may have some unknown values (remaining three rules). A work list algorithm is used in the implementation.

The result of the analysis is used to eliminate field accesses. This is carried out through variable renaming. Indeed, if the predicate $x \hookleftarrow ?$ does not hold and there is a single variable y such that $x \hookleftarrow y$, then all occurrences of variable x can be replaced by variable y. If we happen to have $\mathrm{def}(x) = z[i]$, this will turn the field access into dead code. This only works when the field value has a single known definition. But there is a second case where a variable renaming can be performed. If we have $\mathrm{def}(x) = y[i]$, the predicate $y \hookleftarrow ?$ does not hold and, for all variable z such that $y \hookleftarrow z$, the definition of z is of the shape $[j \,|\, \ldots, t, \ldots]$ where a same variable t is at index i, then all occurrences of variable x can be replaced by variable t.

4.4. Dead code elimination

An analysis is performed to determine which parts of the code are reachable and which variables are used. Unreachable code and effect-free expressions whose results are not used are discarded.

DOI: 10.1002/spe

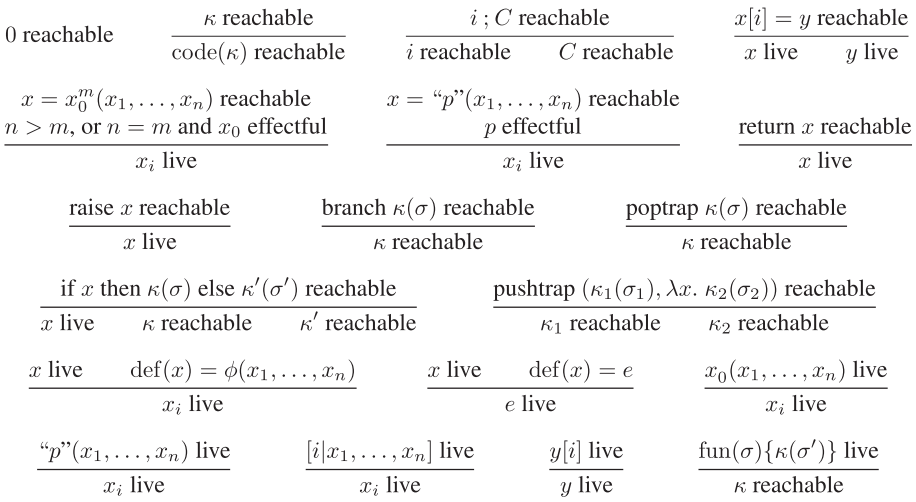


Figure 10. Live variable analysis.

So as to have a more precise result, the compiler first determines which functions may be effectful. We write 'x effectful' to mean that the functions bound to variable x may be effectful. In the current implementation, nonterminating functions are considered effectful; expressions that may only raise exceptions due to programmer errors, such as out-of-bound accesses, are not considered as effectful.

The live variable analysis is specified in Figure 10 as five mutually defined predicates: 'κ reachable', 'C reachable', 'i reachable', 'x live', and 'e live'. Location 0 (the program entry point) is reachable. If a location κ is reachable, then the code of the block at location κ, written code(κ), is reachable. If a piece of code C is reachable, all the instructions it contains are reachable. If a block update or an effectful assignment is reachable, then all its free variables are live. The variable in a reachable return or raise instruction is live. If a control instruction is reachable, then all the locations it points to are reachable. In the case of a reachable conditional instruction, the condition variable is live. If a block parameter is live, then the corresponding arguments are also live. If a live variable is assigned to some expression e, then the expression is live. If an expression is live, then all its free variables are live. Finally, if a closure expression is live, then its body location is reachable. This analysis is implemented using recursive functions.

Then, dead code elimination can take place. Unreachable blocks are discarded. Assignments $x = e$, where x is not live and e is not effectful, are removed. Block parameters that are not live are also removed, as well as the corresponding block arguments.

During the live variable analysis, the compiler actually computes how many time each variable is used. This is only a slight variation of the live variable analysis presented earlier, where a counter, associated to each variable, is incremented each time one would deduce that a variable is live. One just has to be careful to consider expressions only once (two distinct rules apply for effectful expressions). This information is used to perform some inlining locally during code generation (Section 5).

The analysis is effective at eliminating unused functions in modules (Figure 15). It is not powerful enough to deal with functors (modules parameterized by other modules) in the general case. An interprocedural analysis would be needed for that.

4.5. Function inlining

The OCaml bytecode compiler does not perform any inlining. There are thus many opportunities for inlining. At the moment, only functions that are used exactly once are inlined. This is guaranteed to make the code smaller and is simple to implement. In particular, no variable renaming is necessary. This is performed as follows. The block containing the function call is split at the call instruction. The function call is replaced by a branch to the function body. The arguments passed to the block can

DOI: 10.1002/spe

be deduced from the function parameters. Each return instruction in the function body is replaced by a branch to the instruction just after the function call, with a single argument that is the return value of the function.

According to our measurements, inlining has no significant impact on performance. On the other hand, it helps for dead code elimination, as it can expose opportunities for code removal. In particular, the dead code elimination algorithm is not smart enough to eliminate unused functions in a functor (i.e., in a higher-order module). On the other hand, if the body of the functor is inlined, unused functions can be eliminated. Function inlining has also turned out to be a good way to stress the compilation process, and hence shake out bugs in the compiler, as it makes the control flow significantly more complex.

5. JAVASCRIPT GENERATION

The compiler first produces a JavaScript abstract syntax tree, which is then printed. Thus, parentheses and whitespaces do not have to be dealt with during code generation. They are added only, when necessary, when serializing the abstract syntax tree. The use of an abstract syntax tree makes it also possible to perform some peephole optimizations at the JavaScript level.

A naive compilation of our running example (Figure 6) would yield the following JavaScript piece of code:

```
function f(x){
 var b = 0 < x;
 if (b) {
  var t = 2; var y = caml_mul(t,x); return y;
 } else { return x; }
}
var x = 10; var z = f(x);
```

We describe the function body. The code for block at location 68 is generated first. The conditional instruction at the end of the block is translated into a JavaScript conditional statement. The code for blocks at locations 72 and 79 is inserted in the branches. The actual result, assuming that function `f` is not inlined, is the following:

```
function f(x){return 0<x?2*x|0:x;}
var y=f(10);
```

Unnecessary variable assignments are avoided (Section 5.1). A direct multiplication can be performed without loss of information (JavaScript uses IEEE 754 floating point arithmetic). The result is converted back to a 32-bit integer using the construction `e|0` (see Section 6 for details). Finally, a peephole optimization turns statement '`if(e)return e1;else return e2;`' into statement '`return e?e1:e2;`'. (This is only to save space; it does not make any performance difference with current JavaScript engines).

We now detail the code generation process: first how expressions are generated and then how the control flow graph is compiled to JavaScript.

5.1. Generating expressions

To obtain compact code, the compiler produces nested expression when possible, skipping assignments to intermediate variables. It is careful to preserve the order of evaluation. For this, three kinds of expressions are distinguished: expressions that always evaluate to the same result and have no side effect are 'pure' (e.g., an integer addition $x + y$), expressions that have side effects are 'mutators' (e.g., an array update), and expressions that do not have side effects but which may evaluate to different values due to side effects are 'mutable' (e.g., a block access $x[i]$).

Pure and mutable expressions can be reordered freely. The order of mutator expressions has to be preserved. A mutable and a mutator expression cannot be swapped. The compiler provides a way to declare external primitives and specifies their kind.

DOI: 10.1002/spe

When compiling a piece of intermediate code, the compiler keeps a set of pending assignments of a JavaScript expression to a JavaScript variable, together with the kind of each of these expressions. When compiling a statement $x = e$, the JavaScript expression corresponding to expression e is produced first. In doing so, any variable defined in the assignment set is replaced by the corresponding expression (and the assignment is removed from the set). Then, if expression e is a mutator, all the assignments corresponding to mutable or mutator expressions are emitted. If expression e is mutable, any assignment corresponding to a mutator expression is emitted (note that there can be at most one). Finally, if the variable x is used only once, the current assignment is added to the set of pending assignments. Otherwise, the assignment is directly emitted. When the variable x is not used, only the code of the expression is emitted, not the assignment to the variable. At the moment, all pending assignments are flushed before the compilation of a function closure. The compiler could be more precise and flush just the assignments to variables occurring in the closure. The pending assignments are also flushed before emitting any control statement.

5.2. Compiling the control-flow graph

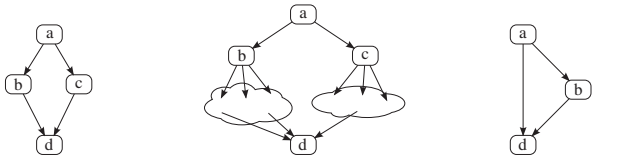
The control flow graph has to be mapped to JavaScript control statements. The compiler uses `for` statements for loops and conditional and switch statements for other control flow graph edges. In both cases, a crucial ingredient is the computation of the 'dominance frontier' [10] of each block. The dominance frontier of a block a is the set of blocks b such that the block a dominates one predecessor of b (every path from the entry that reaches this predecessor has to pass through block a) but not all of them.

The OCaml bytecode compiler always generates reducible control flow graphs: there are no cross edges (edges that point to a block that is neither an ancestor nor a descendant of the current block). Loops can be found by a depth-first traversal: a block is the start of a loop if there exists a back edge pointing to it. Loops are compiled into `for` statements:

`label:for(;;){(`*loop body*`) break;}`

In the loop body, branching back to the beginning of the loop is performed by emitting the code for passing arguments to the block at the top of the loop followed by a `continue` statement. The argument passing must be performed by parallel variable renaming: for instance, when the parameters are a pair of variables (a, b) and the arguments are the same pair in reverse order (b, a), the content of the two variables must be swapped; it would be incorrect to first assign the contents of b to a then the new contents of a to b. When loops are nested, a label is used to specify at the beginning of which loop the execution should proceed. Blocks belonging to the dominance frontier of the first block of the loop are not part of the loop, but right after the loop. If the flow of control reaches the end of the loop body without encountering a backward edge, the execution should leave the loop, hence the `break` instruction at the end of the loop.

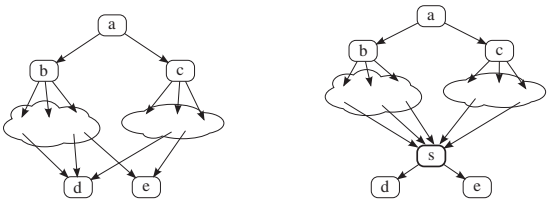
We now explain how forward edges are handled. A recursive process is used, which compiles a block and all the blocks it dominates. We illustrate this process on the graph in Figure 11(a). The compilation starting at block a proceeds as follows. First, the instructions of the block are compiled. Then, the control instruction is handled. Here, there are two edges; hence, a conditional statement is used. With more than two edges, a switch statement is used. If there is a single edge (e.g., as for block b), no code is produced at this point, and the compilation process proceeds linearly right after



(a) simple conditional (b) general shape (c) with critical edge

Figure 11. Recognizing conditionals.

DOI: 10.1002/spe



(a) before (b) after

Figure 12. Switch insertion.

the already-emitted statements. The argument passing code corresponding to each edge is inserted in the corresponding branch of the conditional. Then, if the target block has a single incoming edge, the block (and the blocks it dominates) is recursively processed. Thus, the instructions of block b are inserted in the first branch of the conditional followed by the argument passing code to block d and similarly for the second branch of the conditional. However, the code corresponding to the instructions of block d is, rightly, not inserted in any of the branches, as the block has two incoming edges. Now that the conditional statement has been produced, the blocks that either are on the dominance frontier of one of the branches or are the target of critical edges coming from block a (i.e., edges that are neither the only edges leaving their source block nor the only edges entering their destination block) are considered. (See Figure 11(c) for an example of critical edge, from block a to block d). Here, block d is the only such a block. The compilation thus proceeds recursively at this block. There can be no such block, for instance, if block c ended with a return instruction rather than branching to block d. In this case, the compilation process stops here. There can also be more than one such blocks, as shown in Figure 12(a). Such control graphs may arise because of the compilation of pattern matching and shortcut boolean operators (`&&` and `||`). Then, an intermediate block s is inserted (Figure 12(b)). A fresh variable x is used to indicate which block ought to be executed next. The block s performs a switch on this variable and dispatches to the adequate block (here, either block d or e). Jumping to block d from one of the branches of the conditional is compiled as first performing the argument passing to this block and then setting the variable x appropriately. The insertion of the intermediate block s is performed on the fly. Indeed, it cannot be performed at the intermediate code level, as its successor blocks have incompatible parameters in general.

6. LANGUAGE-SPECIFIC ISSUES

6.1. Deviations from the standard OCaml implementation

At the moment, integers are 32 bits rather than 31 or 63 bits with the standard OCaml implementation depending on the architecture. Indeed, there is no implementation reason to lose one bit. The compiler could provide 31-bit integers at a reasonable cost by masking appropriately the result of each integer operation. But we do not think this choice will result in many compatibility issues. Programs for which the integer size matters already deal with several sizes and usually use masking for that. For instance, the `Random` module from the standard library, which implements pseudo-random number generators, works perfectly well with our compiler.

The compiler performs self-tail call optimization but not general-tail call optimization. The general case could be implemented using trampolines, but at a high cost. Indeed, implementing properly tail call optimization when targeting a runtime with no tail call support was considered both for Scala [11] (JVM) and Hop [12] (JavaScript) but was not implemented for any of these languages. We hope JavaScript interpreters will eventually support tail call optimization in strict mode [2], which does not allow stack inspection.

6.2. Integer operations

JavaScript only provides floating-point arithmetic operations (using double-precision 64-bit format IEEE 754 values). It performs logical operations by first converting the operands to 32-bit integers

DOI: 10.1002/spe

(non-integer values are truncated toward zero). Addition and subtraction on 32-bit integers can be implemented by performing the corresponding float operation and then converting back to integer using a logical operation `(x+y)|0`. Multiplying two 32-bit integers can require up to 62 significant bits (not including the sign), whereas floats only provide 53 significant bits. Thus, a custom multiplication function is used in general. When one of the operands is statically known to be small, thanks to the data flow analysis, a direct JavaScript multiplication is performed, followed by a conversion to 32-bit integer. Division by zero raises an exception in OCaml, whereas it returns `Nan` ('not a number') in JavaScript. Thus, a custom division function is used when the compiler cannot determine that the divisor is different from 0. The modulo operation is implemented similarly.

6.3. Array bound checking

JavaScript returns the `undefined` value when an out-of-bound array access is performed rather than raising an exception. The compiler follows the OCaml semantics and inserts bound checks by default.

6.4. Function invocation

The OCaml language encourages a curried style. From a semantic point of view, OCaml functions always take a single argument. A function expecting more than one argument is actually a function that takes a first argument and returns a function consuming the remaining arguments. This style is inefficient if implemented naively. Thus, implementations of the language use n-ary functions internally, with a suitable invocation strategy to simulate the expected semantics.

The data flow analysis is used to optimize function calls. At each call point, the expected arities of the possible closures are computed. If the arity is known and matches the number of arguments, the function can be called directly. Otherwise, an intermediate function is called with the closure and the arguments to perform the call appropriately. The compiler generates one such function for each call-site arity. The function used when two arguments are provided is given as follows. In JavaScript, the property `f.length` of a function `f` is its expected arity. If the arities correspond, which is likely, the function is invoked directly. Otherwise, the invocation is performed by a generic function `caml_call_gen` that handles currying.

```
function caml_call_2(f,x,y) {
 return f.length==2?f(x,y):
           caml_call_gen(f,[x,y]); }
```

6.5. Strings

JavaScript only provides immutable UTF-16 strings, whereas OCaml strings are mutable arrays of eight-bit characters. Thus, strings are implemented as objects that act as proxies for three possible representations: a UTF-16 JavaScript string, a string of bytes stored in a JavaScript string (1 byte per UTF-16 code unit), and a JavaScript array of bytes. The first representation is used when converting to and from JavaScript. The second allows efficient string read access and concatenation and can be converted efficiently to the first. The last is only used when the string is modified. Conversions are automatically performed when needed.

6.6. Concurrency

JavaScript has no multithreading support. Thus, the OCaml thread library cannot be used. However, the Lwt cooperative thread library [13] works out of the box, without modification nor recompilation.

7. INTEROPERABILITY WITH JAVASCRIPT

It is crucial to be able to access the browser APIs in a natural way. The module `Js` provides functions for manipulating JavaScript values (strings, booleans, and `null` and `undefined` values) from

DOI: 10.1002/spe

OCaml and performing conversions between the two worlds. For instance, the parametric-type `'a Js.opt` stands for values that are either `null` or of type `'a`. A number of functions are available to manipulate values of this type: testing whether a value is `null`, performing different operations whether a value is `null` or not, and so on.

JavaScript object types are encoded using OCaml object types. An abstract-type constructor `Js.t` with a phantom parameter is used to denote JavaScript objects. The parameter describes the methods and properties of the objects. For instance, consider this class-type definition (we show only some of the methods).

```
class type canvasContext = object
 method canvas : canvasElement t readonly_prop
 method save : unit meth
 method restore : unit meth
 method scale : float -> float -> unit meth
 method rotate : float -> unit meth
 method lineWidth : float prop
 ...
end
```

Such a class-type definition does not correspond to any actual OCaml object. It is just used to specify a type abbreviation `canvasContext` that stands for an object type with the given methods. Then, type `canvasContext Js.t` is the type used to denote JavaScript `canvasContext` objects. Each method in the OCaml object type corresponds either to a property or a method of the JavaScript object. We use type constructors with a phantom parameter to differentiate the different cases. Constructor `readonly_prop` is used for read-only properties and constructor `prop` for read-write properties. For instance, property `canvas` is read-only and contains a `canvasElement` JavaScript object; property `lineWidth` contains a floating-point number and can be modified. Field `scale` is a method with two arguments of type `float` and that returns nothing. The result type is marked by a parametric-type `meth`. This delimits precisely the part of the type corresponding to arguments and the part corresponding to the method return value, even when a function is returned. (This is important, as JavaScript does not support currying.) Although JavaScript is untyped, this scheme works well in practice. Indeed, browser APIs are specified using the Document Object Model, which is a typed language-independent convention.

A Camlp4 syntax extension is used to perform method invocations and to access object properties in a type-safe way. For instance, a method invocation is written `e##m(e1,...,en)`, with the number of arguments made explicit by the tuple notation. This method invocation expression is desugared into a call to an untyped primitive `caml_js_meth_call (e,'m',[|e1,...,en|])`, with appropriate type constraints not shown here. The array makes the number of arguments explicit. This primitive is recognized by the compiler that generates a direct JavaScript method call. Object property access `e##m` and update `e##m <- e'` are also defined as syntax extension and are compiled in a similar way.

A naming trick is used to map several method names on the OCaml side to the same method in JavaScript to deal with overloading (in particular, methods with optional parameters are common in JavaScript): any leading underscore character is removed from the OCaml method name; then, the underscore occurring last and whatever follows are also removed. Thus, names `concat`, `_concat`, and `concat_2` are all mapped to the same JavaScript name `concat`.

8. CURRENT STATUS AND PERFORMANCE

8.1. Status

The compiler is publicly available[‡]. It is currently about 6000 lines of OCaml. We believe it is a solid implementation that can be used for real applications. Most of the OCaml standard library is

[‡]http://ocsigen.org/js_of_ocaml/

DOI: 10.1002/spe

supported. A binding for a large part of the browser APIs, including manipulation of the HTML Document Object Model tree, is provided.

We have written a few sample demos that can be tried online. One of them is an OCaml toplevel than runs fully on the browser. It includes both the OCaml compiler and the Js_of_ocaml, compiled to JavaScript; commands are compiled on the browser to bytecode, then to Javascript, before being executed. A second one is an interactive tree viewer, which can be used to browse a large phylogenetic tree of animals, laid out on the hyperbolic plane. A third one is a graphic viewer with both a GTK and Web user interface (a lot of code being shared between the two), which we use to visualize the huge graphs of package dependencies in GNU/Linux distributions. The last one is a 3D real-time animation of the Earth based on the HTML canvas element.

8.2. *Performance*

We tested the compiler with several programs, most of them taken from the standard OCaml benchmarks, and two JavaScript tests rewritten in OCaml: splay from the V8 benchmarking suite and raytrace from the Webkit benchmarking suite. We ran the generated programs with the Google V8 (version 3.1.5), Apple Nitro (revision 79445 of the Subversion repository of Webkit), and Mozilla JaegerMonkey (revision 62992 of the Mercurial repository) JavaScript engines. All tests were performed on an Intel Core i7-870 platform running a 64-bit Debian GNU/Linux operating system. Each program was run at least ten times. We report average time. In each case, the measurement error is below 3% at 99% confidence, as estimated using Student's t-distribution.

The execution time of the code generated by our compiler is compared on Figure 13 with the execution time of the same programs compiled with the OCaml native code compiler (`ocamlopt`) and the OCaml bytecode compiler (`ocamlc`). The running times are normalized, and we take the performance of the Google V8 engine as reference. Overall, the JavaScript engines are faster than the bytecode interpreter, V8 being the fastest. Exceptions are extremely expensive, especially with Nitro: `boyer` and `kb` heavily use exception; variants `boyer_no_exc` and `kb_no_exc` use option types rather than exceptions in most cases and are thus more competitive. Strings and 64-bit integers are not natively supported in JavaScript. This explains the low performance with `hamming` and `splay`. JaegerMonkey appears to be slower than other engines with simple recursive functions (`fib` and `takc`). We were not able to measure its performance on the `hamming` and `splay` benchmarks as the standalone JaegerMonkey engine fails with an 'out of memory' error on these benchmarks (although Firefox 4.0 is able to run them successfully).

8.3. *Comparison with JavaScript programs*

Making a fair comparison with handwritten JavaScript programs is hard. We are unlikely to match the performance of heavily hand-tuned programs. On the other hand, one may hope to match casually written JavaScript code. We hope the following results will give an idea of where we stand, although they should be taken with a grain of salt. We compared the performance of compiled programs and handwritten JavaScript versions of the same programs: `bdd`, `fft`, `fib`, `raytrace`, and `splay` (Figure 14). We translated the first three programs from OCaml and the last two

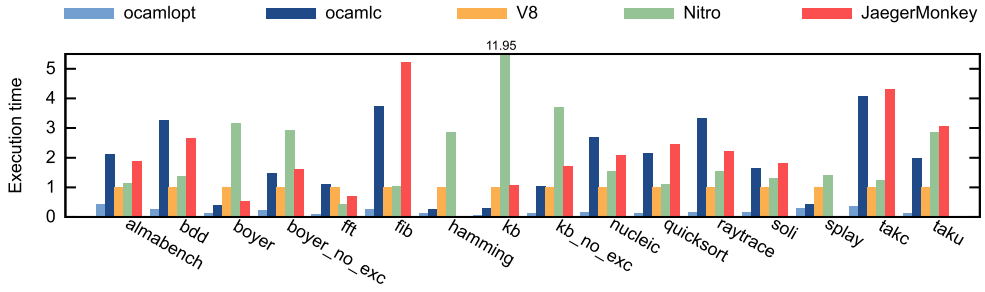


Figure 13. Relative execution time.

DOI: 10.1002/spe

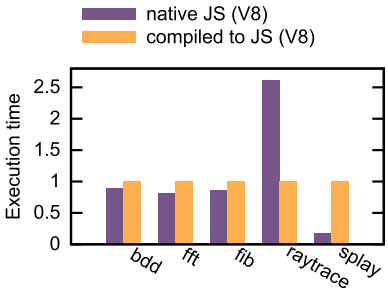


Figure 14. Comparison of the generated code with handwritten JavaScript code.

from JavaScript. In three cases, the generated program is only a little bit slower than the native version. This tends to indicate that the generated code is quite good. The JavaScript version of `raytrace` creates many objects for vectors and colors. This is very natural as it allows to name the vector components. The OCaml version uses records, which are compiled to literal arrays. This appears to be much faster. The `splay` program performs quite many string operations. We are at a disadvantage here, as we cannot use the native JavaScript string implementation.

8.4. Size of generated code

As mentioned already, it is important to produce compact code. The relative size of the OCaml source code and of the bytecode, compared with the code produced by the compiler, is shown in Figure 15. For the latter, we distinguish between generated code and runtime. The runtime consists of handwritten JavaScript functions that correspond to C functions in the bytecode interpreter. It is not optimized for space at the moment. We omitted benchmarks that were too small to give significant results and added several concrete programs: O'Browser [14] examples (`minesweeper`, `sudoku`, and `boulderdash`), an ocamljs [15] example (`canvas`), and some other small programs (`planet` and `cubes`). These programs make more use of external libraries, which explains the large size of the bytecode compared with the source code. We also tested the compiler on large OCaml programs: the OCaml bytecode compiler, the Ocsigen Web server, the Unison file synchronizer, and the compiler itself.

In all cases but one (`boyer`, which contains a large constant value), the generated code is smaller than the bytecode. For a large program, with little dead code, the generated code is about 30% smaller than the bytecode. It is much smaller for medium-sized programs as a large part of the included libraries is dead code.

When comparing source code with compiled code, we should keep in mind that some of the generated codes come from libraries, which are not included in the given source code size. We believe it is also fairer not to take into account the size of runtime code, as this size is bounded and become negligible for large programs. With these caveats, the generated code is smaller than the source code for most benchmarks and comparable for others. For medium-sized applications, it is larger, as the code corresponding to libraries is not accounted for by the source size. Overall, the compiler appears to produce consistently compact code.

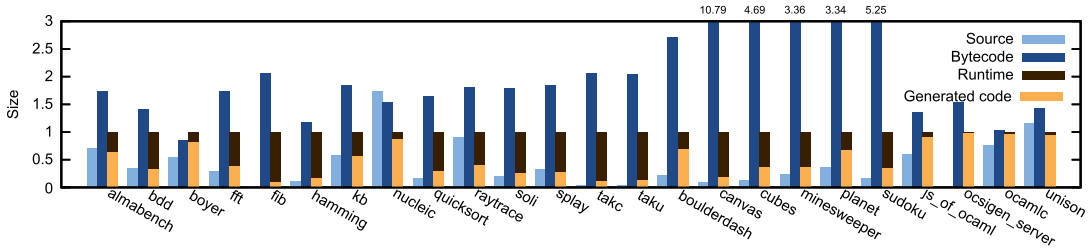


Figure 15. Relative size of OCaml source programs and bytecode with respect to generated programs.

DOI: 10.1002/spe

8.5. *Performance optimizations*

We present in Figure 16 the impact on the execution time (using the V8 engine) of disabling the function call optimization and of disabling bound checks for array and string accesses (option `-unsafe` of the OCaml compiler).

The function call optimization is very effective for programs performing many function calls (`bdd`, `fib`, `quicksort`, `raytrace`, `takc`, and `taku`). Array bound checking has a large impact on programs working on arrays (`fft`, `quicksort`, and `soli`).

8.6. *Code size optimizations*

The code size impact of turning off inlining, dead code elimination, or compact expression generation (Section 5.1) is shown in Figure 17. We compare the size of the generated code, with the runtime code omitted. As expected, dead code elimination is extremely effective on medium-sized programs making use of libraries. Overall, the two other optimizations each yield a code size reduction of about 5%. This is small but not negligible. The improvement is sometimes larger with inlining, as it can expose opportunities for dead code removal.

8.7. *Compilation time*

We report the time taken by our compiler to translate some bytecode programs to JavaScript in Table I. Even large programs are compiled in less than 30 s.

8.8. *Conclusion*

Our benchmarks show that the compiler generates compact code with good performance for most tested programs, even compared with handwritten JavaScript programs. By using compiler options to disable some optimizations, we checked that our analyses were effective, both with regard to performance and code size. Finally, we can see that modern implementations of JavaScript are getting reasonably fast, as the performance achieved is comparable with those of OCaml bytecode programs.

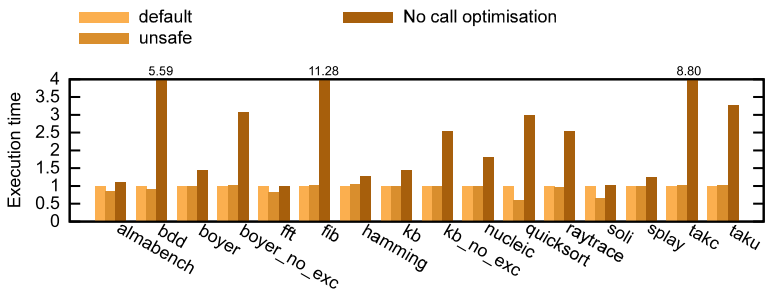


Figure 16. Impact of array bound checking and function call optimization.

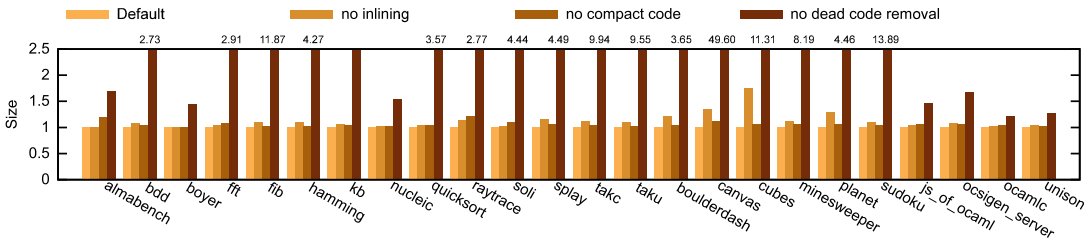


Figure 17. Impact of optimizations on the size of generated code.

DOI: 10.1002/spe

Table I. Compilation times.

Program	Time (s)
almabench	0.06
bdd	0.04
boulderdash	0.54
fib	0.02
ocamlc	9.81
ocsigen_server	15.91
unison	3.82

9. RELATED WORK

The compilation process from a low-level language to a high-level language has clear relationship with decompilation [16–18]. In particular, one finds the same issues of recovering the control flow and mapping it to high-level constructions. However, whereas decompilation put the emphasis on readability, we are rather interested in concise and fast code. In particular, basic techniques are sufficient in our case. We are not interested in choosing informative names for variables or in detecting specific high-level constructions.

A growing number of compilers target JavaScript. We present the most interesting and relevant ones. First, several other compilers for functional languages have been written: the Links compiler [19], SMLtoJs [20], F# Web Tools [21], ocamljs [15], and Hop [12, 22]. Of these compilers, only Hop, which compiles a dialect of Scheme, put the emphasis on performance and is actively maintained at the moment. Although we try to follow closely the behavior of the standard OCaml, the ocamljs compiler takes a more pragmatic approach and attempts to map OCaml data types (functions, strings, objects, and others) into the corresponding JavaScript data types. Because of the semantic differences, this can yield runtime failures that are not caught by the type system. Ocamljs generates much slower and much larger code than our compiler. Programs run typically several times slower. It appears that function calls are implemented in an inefficient way. The compiler does no static analysis on the argument length of a function, and thus, all function calls involve auxiliary functions that manage currying. It implements tail recursion properly, by using trampolines.

The Google Web Toolkit [6] was one of the first toolkit providing a compiler targeting JavaScript and aiming for good performances. It makes it possible to write applications in Java and run them on all browsers. We find it noteworthy that it still only supports Java 6 1 year after the release of Java 7. As a contrast, no change to the Js_of_ocaml compiler were necessary to support OCaml 4.0, a new major release of the language: we only had to implement few additional runtime functions. DART [23] is a language designed by Google that aims to become an alternative to JavaScript on Web browsers. Much effort is currently being put in making the DART compiler generate efficient and compact JavaScript code.

Emscripten [24] is an LLVM-to-JavaScript compiler, taking as input the intermediate representation of the now popular low level virtual machine (LLVM) compilation framework [25]. It makes it possible to run C and C++ code on Web browsers. The performances are quite impressive. Indeed, arithmetic operations and array accesses are especially well optimized by current JavaScript engines, and Emscripten is able to take advantage of that. However, a low-level memory model is used: basically, C programs manipulate an array of integers and cannot directly access any other JavaScript value. As a consequence, some glue code has to be written in JavaScript to access the browser APIs.

XMLVM [26–30] is a cross-compiler toolchain that takes as input Java or .Net bytecode and can produce as output one of these bytecodes, JavaScript codes, or Objective-C codes. This makes it possible to develop Android applications and cross-compile them to other smartphones. At the moment, the compilation is implemented in a very straightforward way, without much optimization. A Javascript bytecode file is first converted into an XML document containing a textual representation of its contents. Then, Javascript code is generated by an XSL stylesheet that

DOI: 10.1002/spe

replaces each Java bytecode instruction by a sequence of JavaScript instructions performing the same operation.

Another approach for running bytecode programs on browsers is to use a bytecode interpreter written in JavaScript. O'Browser [14] shows that this is indeed feasible, at least for short programs. But the performance hit is high.

10. FUTURE WORK

The compiler was started as an experiment: when compiling OCaml bytecode to JavaScript, was it possible to have an acceptable performance and keep the size of the generated code moderate? The result, as shown by the benchmarks, is well beyond our expectations. But there is always room for improvement. In particular, there remains some low-hanging fruits regarding code size. A better choice of variable names, taking into account the scope of variables, could yield a 10% to 15% improvement in code size. This requires a liveness analysis followed by graph coloring, as if performing register allocation [31, 32].

A natural application of the compiler is for multi-tier programming. We have extended the Ocsigen Web programming framework [33] to provide a uniform framework where OCaml is used both on the server and on Web browsers.

We are interested in reusing the front-end of the compiler to target other languages. Native code could be generated through the LLVM compilation framework [25]. This would yield an alternate native code compiler, more portable than the current one. Other possible targets include the Java and .Net virtual machines, or the Dalvik virtual machine, to run OCaml programs on Android. A difficulty with these targets is the generation of typed bytecode from untyped bytecode. Finally, the compiler could also be made to output optimized OCaml bytecode. This would be especially interesting for resource-constrained systems, such as microcontrollers.

A last possible direction would be to take a different source bytecode, such as the Java bytecode or .NET Common Intermediate Language. Compilation of method invocation should be straightforward. On the other hand, more complex analyses should be required for eliminating dead code effectively, as the control flow is usually less explicit than in OCaml.

ACKNOWLEDGEMENTS

This work was performed at the IRILL center for Free Software Research and Innovation in Paris, France, and was supported by the French National Research Agency (ANR), PWD project, grant ANR-09-EMER-009-01.

REFERENCES

1. Leroy X, Doligez D, Garrigue J, Vouillon J, Rémy D. The objective Caml system. Software and documentation available on the Web, 2013. Available from: http://caml.inria.fr/.
2. E C M A International. *ECMA-262: ECMAScript Language Specification*, (3rd edn). ECMA (European Association for Standardizing Information and Communication Systems): Geneva, Switzerland, 1999.
3. Lindholm T, Yellin F. *Java Virtual Machine Specification*, (2nd edn). Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1999.
4. Montelatici R, Chailloux E, Pagano B. Objective Caml on .NET: the OCamIL compiler and toplevel. *3rd International Conference on .NET Technologies*, Pizen, Czech Republic, 2005; 109–120. Available from: http://hal.archives-ouvertes.fr/hal-00003784/en/.
5. Leroy X, Pessaux F. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems* 2000; **22**:340–377. DOI: 10.1145/349214.349230.
6. Google Inc. Google Web Toolkit, 2013. Available from: http://code.google.com/webtoolkit/.
7. Leroy X. The ZINC experiment: an economical implementation of the ML language. *Technical Report 117*, INRIA, 1990.
8. Appel AW. SSA is functional programming. *ACM SIGPLAN Notices* 1998; **33**:17–20. DOI: 10.1145/278283.278285.
9. Aycock J, Horspool RN. Simple generation of static single-assignment form. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*. Springer-Verlag: London, UK, 2000; 110–124.
10. Cooper KD, Harvey TJ, Kennedy K. A simple, fast dominance algorithm. *Software Practice and Experience* 2001; **4**:1–10.

DOI: 10.1002/spe

11. Schinz M, Odersky M. Tail call elimination on the Java virtual machine. *Proceedings of the First Workshop on Multi-Language Infrastructure and Interoperability*, Firenze, Italy, 2001; 158–171.
12. Loitsch F, Serrano M. Hop client-side compilation. In *Trends in Functional Programming, Trends in Functional Programming*, Vol. 8, Morazán MT (ed.). Intellect, UK/The University of Chicago Press: USA, 2008; 141–158.
13. Vouillon J. Lwt: a cooperative thread library. In *Workshop on ML, Victoria, British Columbia, Canada*. ACM: New York, NY, USA, 2008; 3–12, DOI: 10.1145/1411304.1411307.
14. Canou B, Balat V, Chailloux E. O'browser: Objective Caml on browsers. In *ML*, Sumii E (ed.). ACM: New York, NY, USA, 2008; 69–78, DOI: 10.1145/1411304.1411315.
15. Donham J. ocamljs, 2013. Available from: http://jaked.github.com/ocamljs/.
16. Cifuentes C. Reverse compilation techniques. *PhD Thesis*, Queensland University of Technology, July 1994.
17. Miecznikowski J, Hendren LJ. Decompiling Java bytecode: problems, traps and pitfalls. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag: London, UK, 2002; 111–127.
18. Miecznikowski J, Hendren L. Decompiling Java using staged encapsulation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. IEEE Computer Society: Washington, DC, USA, 2001; 368.
19. Cooper E, Lindley S, Wadler P, Yallop J. Links: Web programming without tiers. In *FMCO, Lecture Notes in Computer Science*, Vol. 4709. Springer: Amsterdam, The Netherlands, 2006; 266–296.
20. Elsman M. SMLtoJs: hosting a standard ML compiler in a web browser. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients, PLASTIC '11*. ACM: New York, NY, USA, 2011; 39–48, DOI: 10.1145/2093328.2093336. Available from: http://doi.acm.org/10.1145/2093328.2093336.
21. Petříček T, Syme D. F# Web tools: Rich client/server web applications in F#, 2007. Unpublished draft.
22. Serrano M, Gallesio E, Loitsch F. Hop: a language for programming the Web 2.0. In *OOPSLA Companion*, Tarr PL, Cook WR (eds). ACM: New York, NY, USA, 2006; 975–985, DOI: 10.1145/1176617.1176756.
23. Google Inc. Dart, 2013. Available from: http://www.dartlang.org/.
24. Zakai A. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH '11*. ACM: New York, NY, USA, 2011; 301–312, DOI: 10.1145/2048147.2048224. Available from: http://doi.acm.org/10.1145/2048147.2048224.
25. Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, 2004; 75–86.
26. Puder A. Cross-compiling Android applications to the iPhone. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ '10*. ACM: New York, NY, USA, 2010; 69–77, DOI: 10.1145/1852761.1852772. Available from: http://doi.acm.org/10.1145/1852761.1852772.
27. Puder A, Haeberling S, Todtenhoefer R. An MDA approach to byte code level cross-compilation. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD '08*. IEEE Computer Society: Washington, DC, USA, 2008; 251–256, DOI: 10.1109/SNPD.2008.109. Available from: http://dx.doi.org/10.1109/SNPD.2008.109.
28. Puder A. Byte code transformations using XSL stylesheets. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD '08*. IEEE Computer Society: Washington, DC, USA, 2008; 563–568, DOI: 10.1109/SNPD.2008.164. Available from: http://dx.doi.org/10.1109/SNPD.2008.164.
29. Puder A, Lee J. Towards an XML-based bytecode level transformation framework. *Electronic Notes in theoretical Computer Science* 2009; **253**(5):97–111. DOI: 10.1016/j.entcs.2009.11.017. Available from: http://dx.doi.org/10.1016/j.entcs.2009.11.017.
30. Puder A, Häberling S. Byte code level cross-compilation for developing Web applications. *Science of Computer Programming* 2009; **74**(5-6):379–396. DOI: 10.1016/j.scico.2009.01.009. Available from: http://dx.doi.org/10.1016/j.scico.2009.01.009.
31. Park J, Moon SM. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems* 2004; **26**(4):735–765. DOI: 10.1145/1011508.1011512.
32. George L, Appel AW. Iterated register coalescing. *POPL*, St. Petersburg Beach, Florida, 1996; 208–218, DOI: 10.1145/237721.237777.
33. Balat V, Vouillon J, Yakobowski B. Experience report: Ocsigen, a Web programming framework. In *ICFP*. ACM: New York, NY, USA, 2009; 311–316, DOI: 10.1145/1596550.1596595.

DOI: 10.1002/spe