From Bytecode to Javascript: the Js_of_ocaml Compiler

Jérôme Vouillon
PPS, University Paris Diderot and CNRS
jerome.vouillon@pps.jussieu.fr

Vincent Balat
PPS, University Paris Diderot and INRIA
vincent.balat@pps.jussieu.fr

Abstract

We present the design and implementation of a compiler from OCaml bytecode to Javascript. We believe that taking bytecode as 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: 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.

Introduction

We present a compiler translating OCaml [16] bytecode into Javascript [9]. This compiler makes it possible to program client-side interactive Web applications in OCaml.

Javascript is the only language that is directly available in most Web browsers and that provides a direct access to browser APIs. (The Flash and Silverlight platforms are not as widely available nor as integrated.) It is thus the mandatory language for developing Web applications. However, one should 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 feed-back to the user. Having to maintain two different pieces of code in two different languages would be error-prone. Concretely, 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. When using a single language, the impedance mismatch in client-server communication is greatly reduced: data still have to be marshalled; but the same type definitions can be used on both side, with no need for translation. Thanks to recent work on highly-optimized JIT-based interpreters, Javascript now exhibits decent performance. For all these reasons, it is a sensible target for a compiler.

We have chosen to take OCaml bytecode as input, rather than 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 (Java Virtual Machine [17]) and .NET Common Language Runtime hardly change, while the source languages continue to evolve. The same is true for the OCaml virtual machine. Thus, 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 is implemented as patches against the main compiler). This is crucial for us. We have seen too many interesting OCaml-related projects die due to lack of maintenance: OCamlIl [21] (a compiler to .NET), OCamlExc [15] (a static analyzer of spurious exceptions), ocamldefun (a defunctorizer), ... 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, though 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 [12] which can be use with Java programs but not Scala programs.

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 datastructures 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 easily use the available Javascript APIs, though 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 tradeoff.

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, but 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 datastructures and how they are represented in Javascript (Section 1). Bytecode programs are converted to an SSA-based intermediate form (Section 2). Some optimizations are performed on the intermediate code (Section 3). Then, the intermediate code is translated to Javascript (Section 4). We document some OCaml-specific issues in Section 5. The compiler would not be usable without ways of manipulating Javascript values and accessing browser

APIs. We deal with this interoperability issue in Section 6. Finally, we have performed extensive benchmarks of the compiler to assess its performance (Section 7).

1. Data Representation

Data representation is crucial performance-wise. One must choose representations that match the OCaml semantics, that can be translated from Javascript, and that are implemented efficiently by Javascript engines.

OCaml has a number of predefined types: integers, floating-point numbers, characters, strings, booleans, 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
```

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 [14] 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 informations 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 represented as integers. Arrays are represented as structured blocks. String and floating-point numbers are stored into unstructured blocks. Functional values are represented as flat closures, that is, blocks with a special tag and that contain a pointer to the code of the function as well as the values of the free variables of the function. Type constructors with no argument, such as `Leaf`, are represented as integers. Other type constructors are represented as 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 represented as memory blocks; functors (that is, higher-order modules) are represented as function 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 represented as Javascript numbers. Structured blocks are represented as Javascript arrays. The first element of the array is the tag; subsequent elements are the contents of the block. Closures are represented as Javascript functions. We take advantage of the scoping mechanism of Javascript. Thus, 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 8-bit characters, while Javascript strings are immutable UTF-16 strings. More details on how integers and strings are handled are provided in Section 5.

We do not perform any special mapping for exceptions; in particular, OCaml exceptions remain generative. OCaml objects are mostly compiled away during bytecode generation. Thus, we do not have to deal with them in a special way. There are just a few bytecode instructions for method resolution which are implemented as Javascript functions.

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

2. From Bytecode to Intermediate Code

2.1 OCaml Bytecode

The OCaml virtual machine [14] 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 2.3 when presenting the conversion from bytecode to intermediate code. In the actual bytecode, the instruction `BGEINT` only takes one target address `l`. The second address `l'` 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 below. 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)
```

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. 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

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

Figure 3. Intermediate Code

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 pop 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.

2.2 Intermediate Code

A variant of static single-assignment form (SSA) is used as 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 the source block to the destination block. As we shall see at the beginning of Section 3, 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 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 l, the block parameters σ and some code C. 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 $x_1, \dots, x_n$. 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 2.4) and deal with memory blocks (Section 3.3).

2.3 Translation to Intermediate Code

The bytecode program is decomposed 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 recursively 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 2.4 for the later). The end of a block can be explicit, right after 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 need its static environment. More precisely, one need 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. Values flow from previous blocks through the accumulator and the stack. We represent symbolically the contents of the accumulator at the beginning of the block as a fresh variable x, and the contents of the current stack frame as a sequence of fresh variables σ. The parameters of the block are then the concatenation x, σ. The translation of a block is specified in Figure 4 using inductive rules. It is defined as a predicate $x\,;\sigma \vdash B \rightsquigarrow C$ where the variable x and the sequence of variables σ represents symbolically the contents of the accumulator and of the current stack frame, 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 [1] and [2], 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 afterwards the value of the accumulator is the top element of the stack. The `BRANCH l` instruction is compiled to a branch instruction. The target block of the branch is at location `l`. 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. Afterwards, 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 `l`. The arguments are expected on the stack. The contents of the accumulator is unknown at the beginning of a function body and will not be used. Hence, a dummy variable z_0 is used to symbolize this contents. The closure is stored in a fresh variable y. Afterwards, the accumulator contains the closure. The `RETURN` instruction is compiled to a corresponding return instruc-

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

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

$$\frac{y \text{ fresh}}{x\,;\sigma \vdash \texttt{BGEINT n,l,l'}\,; B \rightsquigarrow y = \texttt{n} \geq x\,; \text{if } y \text{ then } \texttt{l}(x,\sigma) \text{ else } \texttt{l}'(x,\sigma)} \qquad \frac{z\,;\sigma \vdash B \rightsquigarrow C \qquad z \text{ fresh}}{x\,;y,\sigma \vdash \texttt{APPLY1}\,; B \rightsquigarrow z = x(y)\,; C}$$

$$\frac{y\,;\sigma \vdash B \rightsquigarrow C \qquad y \text{ and } \sigma' \text{ fresh} \qquad |\sigma'| = \text{arity}(\texttt{l})}{x\,;\sigma \vdash \texttt{CLOSURE 0,l}\,; B \rightsquigarrow y = \text{fun}(\sigma')\{\texttt{l}(z_0,\sigma')\}\,; C} \qquad x\,;\sigma \vdash \texttt{RETURN n}\,; B \rightsquigarrow \text{return } x \qquad x\,;\sigma \vdash \texttt{STOP} \rightsquigarrow \text{stop}$$

Figure 4. Translation to intermediate code

tion. The number `n` of stack items to be discarded is not useful after translation. The returned value x is in the accumulator.

When compiling function, one actually needs to deal with the environment of the current function, that contains the values of the free variables of the function. We only sketch the translation of functions with non-empty 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,l` 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 $x\,;\sigma\,;\eta \vdash B \rightsquigarrow C$ actually 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 similar to the `ACCn` instructions, except that the new contents of the accumulator is taken from the environment rather than from the stack.

$$\frac{y\,;\sigma\,;\eta \vdash B \rightsquigarrow C \qquad \eta(\texttt{n}) = y}{x\,;\sigma\,;\eta \vdash \texttt{ENVACC n}\,; B \rightsquigarrow C}$$

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 that 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 • for an empty sequence of variables).

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`	$g = \text{“}{*}\text{”}(f,e)$
$g\,;e$	`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`	$k = \text{fun}(l)\{68(z,l)\}$
$k\,;\bullet$	`85 PUSH`	
$k\,;k$	`86 CONSTINT 10`	$m = 10$
$m\,;k$	`88 PUSH`	
$m\,;m,k$	`89 ACC1`	
$k\,;m,k$	`90 APPLY1`	$n = k(m)$
$n\,;k$	...	

Figure 5. Example of translation

Besides, an exception handler $(x,(l_2,\sigma))$ 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 as well as 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.

2.4 Exceptions

Three bytecode instructions are dedicated to exception handling.

$$I \quad ::= \quad \ldots \mid \texttt{RAISE} \mid \texttt{PUSHTRAP l} \mid \texttt{POPTRAP}$$

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 pop 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.

$$\begin{array}{rcl} c & ::= & \text{pushtrap } (l_1,\sigma_1),(x,(l_2,\sigma_2)) \\ & \mid & \text{poptrap } (l,\sigma) \ \mid \ \text{raise } x \ \mid \ \ldots \end{array}$$

3. Code Analyses and Transformations

Several code analyses and transformations are performed on the intermediate code in order 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, that does not support currying. To deal with this, the closures possibly involved at each call point are computed (Section 3.3). Then, optimized function calls are generated when the closure arities match the number of arguments provided (see Section 5 for details). Self-tail call are very common and

```
block 68()
  $c = 0 \geq l$
  if $c$ then 79() else 72()

block 72()
  $f = 2$
  $g =$ “*”$(f, l)$
  return $g$

block 79()
  return $l$

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

Figure 6. Code after redundant variable removal

should be implemented using constant stack depth (Section 3.1). Finally, a difficulty for dead code elimination (Section 3.4) 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 their contents (Section 3.3).

At the moment, the compiler only uses intraprocedural analyses. Indeed, they are much simpler to implement and are already quite effective. In order 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 to 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, $\phi(x, x) = \phi(x)$.

3.1 Self-Tail Call Optimization

Tail recursion optimization is performed first. Indeed, this optimization changes the program control flow and thus cannot be easily applied after the transformation in Section 3.2.

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:

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

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:

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

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.

3.2 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 [2]. Whenever one has $\mathrm{def}(x) = \phi(y)$ or $\mathrm{def}(x) = \phi(x, y)$, one can replace all occurrences of variable x by variable y. Intuitively, any value assigned to variable x must in both cases have been assigned to variable y beforehand. By performing this renaming repeatedly until no such

$$\frac{\mathrm{def}(x) = \phi(x_1, \ldots, x_n) \qquad x_i \hookleftarrow y}{x \hookleftarrow y}$$

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

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

Figure 7. Propagation of known values

definition remains, one reduces the number of ϕ-functions (that is, 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 [2], which the OCaml compiler always generates. The implementation does not actually perform variable substitution eagerly. A union-find datastructure is used to keep track of which variable should be replaced by which. A global substitution is performed once no simplification is possible anymore. The block parameter and argument simplification is also not performed at this point, but by dead code elimination (Section 3.4). 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.

3.3 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 5), integer multiplication (Section 5), Javascript method invocations (Section 6), ...

The analysis deals in a special way with the following memory block operations. 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.

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

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

The first step of the analysis is the computation of predicate $x \hookleftarrow y$, as specified in Figure 7. The idea is to track how values

$$\frac{\begin{array}{c}\operatorname{def}(x) = y(\ldots, z, \ldots)\\ z \mapsfrom t\end{array}}{t \text{ escapes}} \qquad \frac{\begin{array}{c}\operatorname{def}(x) = \text{“}p\text{”}(\ldots, z, \ldots)\\ z \mapsfrom t\end{array}}{t \text{ escapes}}$$

$$\frac{\text{return } x \in \mathcal{P} \qquad x \mapsfrom y}{y \text{ escapes}} \qquad \frac{\text{raise } x \in \mathcal{P} \qquad x \mapsfrom y}{y \text{ escapes}}$$

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

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

Figure 8. Escaping values

$$\frac{\begin{array}{c}\operatorname{def}(x) = \phi(x_1, \ldots, x_n)\\ x_i \mapsfrom ?\end{array}}{x \mapsfrom ?} \qquad \frac{\begin{array}{c}\operatorname{def}(x) = y[i]\\ y \mapsfrom ?\end{array}}{x \mapsfrom ?}$$

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

$$\frac{\operatorname{def}(x) = y[i] \qquad y \mapsfrom z \qquad z \text{ mutable}}{x \mapsfrom ?}$$

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

Figure 9. Propagation of unknown values

flow through code blocks and memory blocks. Basically, source variables are propagated through block parameters (fist rule) and field accesses (second rule). In all other cases, we take $x \mapsfrom x$. The implementation collects for each variable x the set of variables y such that $x \mapsfrom y$. A standard work list algorithm is used. A pitfall is that the dependency graph indicating when the value of a variable should be recomputed is not static. Indeed, for the second rule, whenever one learns that $y \mapsfrom z$, 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 anywhere in 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 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 $z \mapsfrom t$, escape (four first rules). If a block escapes, then the values of all its fields also escape (fifth 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 $x \mapsfrom ?$ specified in Figure 9 is computed. It indicates which variables x may contain other unknown values besides the values given by predicate $x \mapsfrom y$. 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 get some unknown values (three remaining rules). A work list algorithm is used in the implementation.

The result of the analysis is used to eliminate field accesses. This is done through variable renaming. Indeed, if the predicate $x \mapsfrom ?$ does not hold and there is a single variable y such that $x \mapsfrom y$, then all occurrences of variable x can be replaced by variable y. If we happen to have $\operatorname{def}(x) = z[i]$, this will turn the field access into dead code. As this only works when the field value has a single known definition, we consider a second case where variable renaming is performed. If we have $\operatorname{def}(x) = y[i]$, the predicate $y \mapsfrom ?$ does not hold and, for all variables z such that $y \mapsfrom z$, 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.

3.4 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.

So as to get a more precise result, the compiler first determines which functions may be effectful. We write ‘x effectful’ to mean that the functions bound to variables x may be effectful. In the current implementation, non-terminating functions are considered effectful; expressions which 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: ‘l reachable’, ‘C reachable’, ‘i reachable’, ‘x live’ and ‘e live’. Location 0 (the program entry point) is reachable. If a location l is reachable, then the code of the block at location l, written $\operatorname{code}(l)$, is reachable. If a piece of code C is reachable, all the instruction it contains are reachable. If a control instruction is reachable, then all the locations it points to 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. 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 analysis, the compiler actually counts how many time each variable is used. This is only a slight variation of the live variable analysis presented here, 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 applies for effectful expressions). This information is used during code generation (Section 4).

The analysis is effective at eliminating unused functions in modules. 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.

3.5 Function Inlining

The OCaml bytecode compiler does not perform any inlining. There are thus lots of 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

$$
0 \text{ reachable} \qquad
\frac{l \text{ reachable}}{\mathrm{code}(l) \text{ reachable}} \qquad
\frac{i\,;C \text{ reachable}}{i \text{ reachable} \qquad C \text{ reachable}} \qquad
\frac{x[i] = y \text{ reachable}}{x \text{ live} \qquad y \text{ live}} \qquad
\frac{\begin{array}{c} x = x_0(x_1, \ldots, x_n)^m \text{ reachable} \\ n > m, \text{ or } n = m \text{ and } x_0 \text{ effectful} \end{array}}{x_i \text{ live}}
$$

$$
\frac{\begin{array}{c} x = \text{“}p\text{”}(x_1, \ldots, x_n) \text{ reachable} \\ p \text{ effectful} \end{array}}{x_i \text{ live}} \qquad
\frac{\text{return } x \text{ reachable}}{x \text{ live}} \qquad
\frac{\text{raise } x \text{ reachable}}{x \text{ live}} \qquad
\frac{\text{branch } (l, \sigma) \text{ reachable}}{l \text{ reachable}} \qquad
\frac{\text{poptrap } (l, \sigma) \text{ reachable}}{l \text{ reachable}}
$$

$$
\frac{\text{if } x \text{ then } (l, \sigma) \text{ else } (l', \sigma') \text{ reachable}}{x \text{ live} \qquad l \text{ reachable} \qquad l' \text{ reachable}} \qquad
\frac{\text{pushtrap } (l_1, \sigma_1), x, (l_2, \sigma_2), l_3 \text{ reachable}}{l_1 \text{ reachable} \qquad l_2 \text{ reachable}} \qquad
\frac{x \text{ live} \qquad \mathrm{def}(x) = \phi(x_1, \ldots, x_n)}{x_i \text{ live}}
$$

$$
\frac{x \text{ live} \qquad \mathrm{def}(x) = e}{e \text{ live}} \qquad
\frac{x_0(x_1, \ldots, x_n) \text{ live}}{x_i \text{ live}} \qquad
\frac{\text{“}p\text{”}(x_1, \ldots, x_n) \text{ live}}{x_i \text{ live}} \qquad
\frac{[i|x_1, \ldots, x_n] \text{ live}}{x_i \text{ live}} \qquad
\frac{y[i] \text{ live}}{y \text{ live}} \qquad
\frac{\mathrm{fun}(\sigma)\{l(\sigma')\} \text{ live}}{l \text{ reachable}}
$$

Figure 10. Live variable analysis

instruction. The function call is replaced by a branch to the function body. The arguments passed to the block can 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 which 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 exposes opportunities for code removal. In particular, the dead code elimination algorithm is not smart enough to eliminate unused functions in a functor (that is, 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 compilers, as it makes the control flow significantly more complex.

4. 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 is 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 4.1). A direct multiplication can be performed without overflow (Javascript uses floating point arithmetic). The result is converted back to a 32-bit integer using the construction `e|0` (see Section 5 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.

4.1 Generating Expressions

In order to get 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* (for instance, integer addition $x + y$); expressions that have side-effects are *mutators*; (for instance, array update); expressions that do not have side-effects but which may evaluate to different values due to side-effect are *mutable* (for instance, 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 specify their kind.

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 expressions 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, the definition of a closure flushes all pending assignments. 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.

4.2 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* [7] of each block. The dominance frontier of a block a is the set of blocks b such that the block a dominates one predecessor

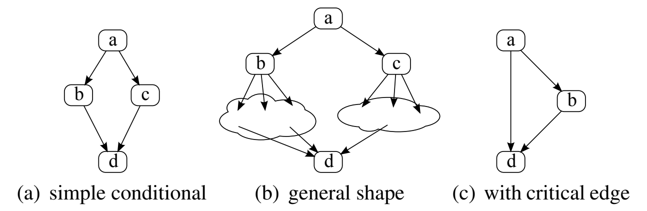
(a) simple conditional (b) general shape (c) with critical edge

Figure 11. Recognizing conditionals

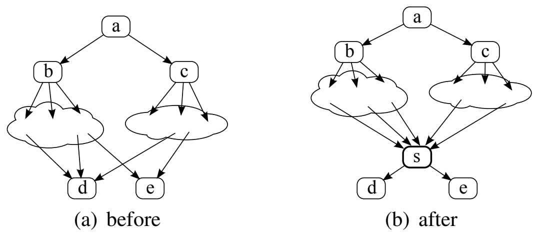
(a) before (b) after

Figure 12. Switch insertion

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 points to a block which 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 proceeds. Blocks in the dominance frontier of the first block of the loop are not part of the loop, but just after. 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 above.

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 follow. 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 (as for block *b*, for instance), no code is produced at this point and the compilation process proceeds linearly right after the already emitted statements (see just below). 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 are either on the dominance frontier of one of the branches or are the target of critical edges coming from block a (that is, edges which 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 due to compilation of pattern matching and shortcut boolean operators (`&&` and `||`). Then, an intermediate block s is inserted (Figure 12(b)). The block s performs a switch on some variable x 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 done on the fly. Indeed, it cannot be performed at the intermediate code level, as its successor blocks have incompatible parameters in general.

5. Language Specific Issues

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 [24] (JVM) and Hop [18] (Javascript), but was not implemented for any of these languages. We hope Javascript interpreters will eventually support tail call optimization in strict mode [9], which does not allow stack inspection.

Integer operations. Javascript only provides floating point arithmetic operations (using double-precision 64-bit format IEEE 754 values). Logical operations are performed by first converting the operands to 32-bit integers (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), while 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, while 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.

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.

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 is 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 below. 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` which handles currying.

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

Strings. Javascript only provides immutable UTF-16 strings, while OCaml strings are mutable arrays of 8-bit characters. Thus, strings are implemented as an object that acts as a proxy for three possible representations: a UTF-16 Javascript string, a string of bytes stored in a Javascript string (one 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.

Concurrency. Javascript has no multi-threading support. Thus, the OCaml thread library cannot be used. However, the Lwt cooperative thread library [26] works out of the box, without modification nor recompilation.

6. Interoperability with Javascript

It is crucial to be able to access the browser APIs in a natural way. A short library provides functions for manipulating Javascript values (strings, booleans, `null` and `undefined` values) from OCaml and performing conversions between the two worlds. For instance, the parametric type `'a opt` stands for values which 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, ...

Javascript object types are encoded using OCaml object types. An abstract type constructor `Js.t` with a phantom parameter is used to denotes Javascript objects. The parameter describes the methods and properties of the objects. For instance, consider the class type definition sketched below (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.) Though Javascript is untyped, this scheme works well in practice. Indeed, browser APIs are specified using the Document Object Model (DOM), 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 a 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 which 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 is also removed. Thus, names `_concat`, `_concat`, and `concat_2` are all mapped to the same Javascript name `concat`.

7. Current Status and Performance

Status. The compiler is publicly available[1]. 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 supported. A binding for a large part of the browser APIs, including manipulation of the HTML DOM tree, is provided.

We have written a few sample demos that can be tried online. One is a 3D real-time animation of the Earth based on the HTML canvas element. Another is a graphic viewer with both a GTK and Web user interface, (lot of code being shared between the two), which we use to visualize the huge graphs of package dependencies in GNU/Linux distributions.

Performance. We tested the compiler with several programs, most of them taken from the standard OCaml benchmarks, as well as 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

[1] `http://ocsigen.org/js_of_ocaml/`

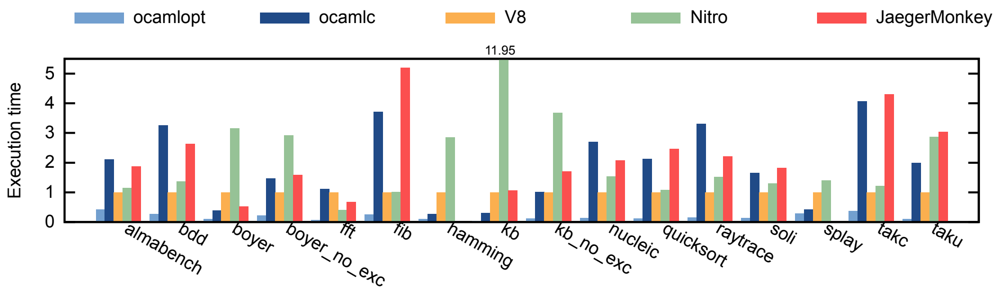

Figure 13. Relative execution time

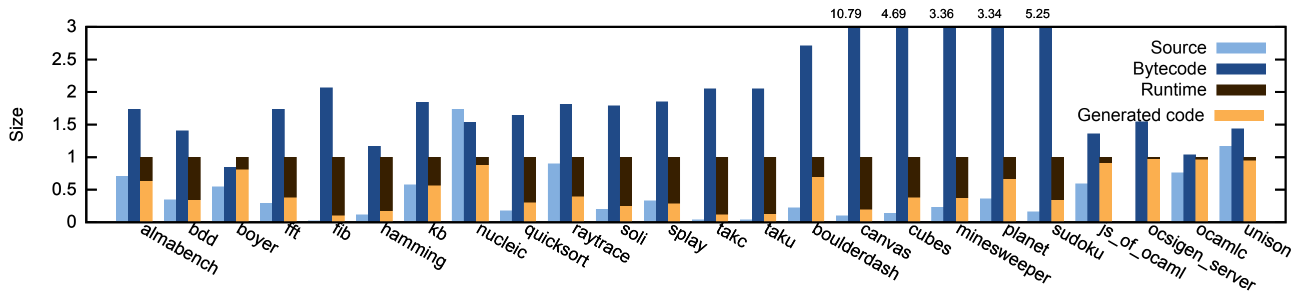

Figure 14. Relative size of OCaml source programs and bytecode w.r.t. generated programs

average time. In each case, the measurement error is below 3% at 99% confidence, as estimated using Students 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 bytecode compiler (`ocamlc`) and the Ocaml native code compiler (`ocamlopt`). 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 expansive, especially with Nitro: `boyer` and `kb` heavily uses exception; variants `boyer_no_exc` and `kb_no_exc` uses 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 (though Firefox 4.0 is able to run the successfully).

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, though 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 15). We translated the first three programs from OCaml and the last two 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 a lot of objects for vectors and colors. The compiler uses literal arrays, which appears to be much faster. The splay program performs quite a lot of string operations. We are at a disadvantage here, as we cannot use the native Javascript string implementation.

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 to the code produced by the compiler, is shown in Figure 14. 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 [4] examples (minesweeper, sudoku and boulderdash), an ocamljs [8] example (canvas) and some other small programs (planet, cubes). These programs make more use of external libraries, which explains the large size of the bytecode compared to 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`, that contains a large constant value), the generated code is smaller than the bytecode. For large program, with little dead code, the generated code is about 30% smaller than 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 code comes from libraries, which are not included for 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 negligeable 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.

Performance optimizations. We present in Figure 16 the impact on execution time (using the V8 engine) of disabling the function

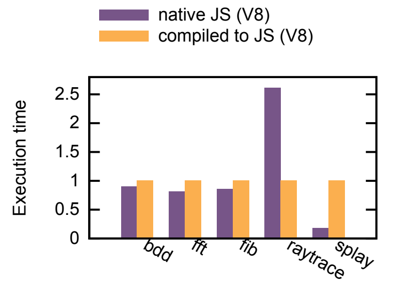

Figure 15. Comparison of generated code with handwritten Javascript code

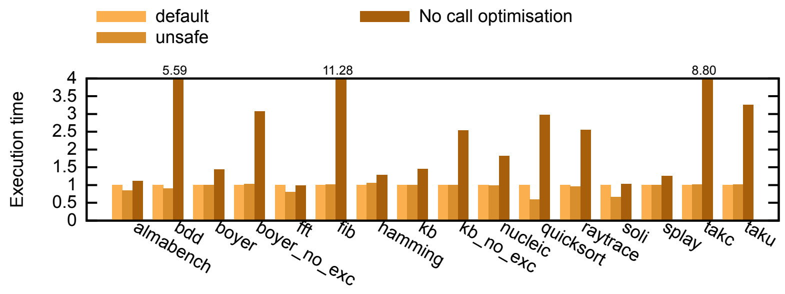

Figure 16. Impact of array bound checking and function call optimization

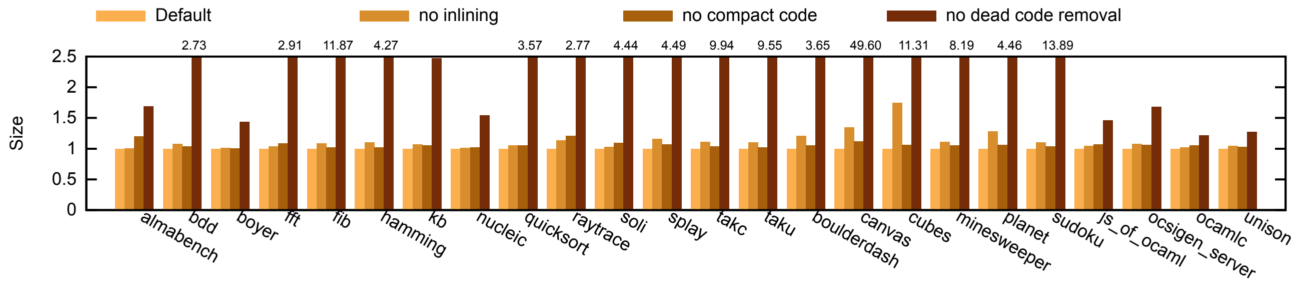

Figure 17. Impact of optimizations on the size of generated code

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

Table 1. Compilation times

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 a lot of function calls (`bdd`, `fib`, `quicksort`, `raytrace`, `takc`, `taku`). Array bound checking has a large impact on programs working on arrays (`fft`, `quicksort`, `soli`).

Code size optimizations. The code size impact of turning off inlining, dead code elimination or compact expression generation (Section 4.1) is shown in Figure 17. We compare the size of the generated code, 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.

Compilation time. We report the time taken by our compiler to translate some bytecode programs to Javascript in Table 1. Even large programs are compiled in less than thirty seconds.

Conclusion. Our benchmarks show that the compiler generates compact code with good performance for most tested programs, even compared to 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 to those of OCaml bytecode programs.

8. Related Work

The compilation process from a low level language to a high level language has clear relationship with decompilation [5, 19, 20]. In particular, one finds the same issues of recovering the control flow and mapping it to high level constructions. However, while 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.

There are many other compilers targeting Javascript including Links [6], SMLtoJs [10], F# Web Tools [23], ocamljs [8], Hop [18, 25], and the Google Web Toolkit (GWT) [12]. All these are full compilers, from a high-level language to Javascript. Only the last two put the emphasis on performance, with GWT also generating compact code.

While we try to follow closely the behavior of the standard OCaml, the ocamljs compiler takes a more pragmatic approach and attempt to map OCaml datatypes (functions, strings, objects, ...) into the corresponding Javascript datatypes. Due to the semantics differences, this can yield runtime failures which are not caught by the type system. Ocamljs generates much slower and much larger code then our compiler. Programs run typically several times slower. It appears that function calls are implemented in a very unefficient way. One of the reason is that Ocamljs uses trampolines to implement tail recursion properly.

We have unsuccessfully attempted to assess the performances of program compiled with the SMLtoJs [10] compiler. It fails to compile with MLton 20100608, the latest release of MLton. We were able to compile it with an older release. But it appears to generate incorrect Javascript code for many benchmarks (such as `boyer.sml`) of the SML/NJ benchmark suite: the code attempts to refer to unbound Javascript variables. This issue appears to be present in both SMLtoJs versions 4.3.5 and 4.3.4.

Another approach for running bytecode programs on browsers is to use a bytecode interpreter written in Javascript. O'Browser [4] shows that this is indeed feasible, at least for short programs. But the performance hit is high.

9. Future Work

The compiler was started as an experiment: when compiling OCaml bytecode to Javascript, was it possible to get acceptable performance as well as 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 [11, 22].

A natural application of the compiler is for multi-tier programming. It would be interesting to extend the Ocsigen Web programming framework [3] to provide a uniform framework where OCaml is used both on the server and on Web browsers. The challenge here is to find a natural way to write programs that run on both sides.

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 [13]. 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.

References

[1] A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33: 17–20, April 1998. ISSN 0362-1340. doi: 10.1145/278283.278285.

[2] J. Aycock and R. N. Horspool. Simple generation of static single-assignment form. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 110–124, London, UK, 2000. Springer-Verlag. ISBN 3-540-67263-X.

[3] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a Web programming framework. In *ICFP*, pages 311–316. ACM, 2009. ISBN 978-1-60558-332-7. doi: 10.1145/1596550.1596595.

[4] B. Canou, V. Balat, and E. Chailloux. O'browser: Objective Caml on browsers. In E. Sumii, editor, *ML*, pages 69–78. ACM, 2008. ISBN 978-1-60558-062-3. doi: 10.1145/1411304.1411315.

[5] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, July 1994.

[6] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006. ISBN 978-3-540-74791-8.

[7] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice and Experience*, 4:1–10, 2001.

[8] J. Donham. ocamljs, 2007. `http://jaked.github.com/ocamljs/`.

[9] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[10] M. Elsman. SMLtoJs, 2007. `http://www.itu.dk/people/mael/smltojs/`.

[11] L. George and A. W. Appel. Iterated register coalescing. In *POPL*, pages 208–218, 1996. doi: 10.1145/237721.237777.

[12] Google Inc. Google Web Toolkit, 2006. `http://code.google.com/webtoolkit/`.

[13] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[14] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[15] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22:340–377, March 2000. ISSN 0164-0925. doi: 10.1145/349214.349230.

[16] X. Leroy, D. Doligez, J. Garrigue, J. Vouillon, and D. Rémy. The Objective Caml system. Software and documentation available on the Web, 2008. URL `http://caml.inria.fr/`.

[17] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

[18] F. Loitsch and M. Serrano. Hop client-side compilation. In M. T. Morazán, editor, *Trends in Functional Programming*, volume 8 of *Trends in Functional Programming*, pages 141–158. Intellect, UK/The University of Chicago Press, USA, 2008. ISBN 978-1-84150-196-3.

[19] J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 368, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1303-4.

[20] J. Miecznikowski and L. J. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 111–127, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4.

[21] R. Montelatici, E. Chailloux, and B. Pagano. Objective Caml on .NET: the OCamIL compiler and toplevel. In *3rd International Conference on .NET Technologies*, pages 109–120, 2005. URL `http://hal.archives-ouvertes.fr/hal-00003784/en/`.

[22] J. Park and S.-M. Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004. ISSN 0164-0925. doi: 10.1145/1011508.1011512.

[23] T. Petříček and D. Syme. F# Web tools: Rich client/server web applications in F#. Unpublished draft, 2007.

[24] M. Schinz and M. Odersky. Tail call elimination on the Java virtual machine. In *Proceedings of the First Workshop on Multi-Language Infrastructure and Interoperability*, 2001.

[25] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the Web 2.0. In P. L. Tarr and W. R. Cook, editors, *OOPSLA Companion*, pages 975–985. ACM, 2006. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176756.

[26] J. Vouillon. Lwt: a cooperative thread library. In *Workshop on ML*, pages 3–12. ACM, 2008. doi: 10.1145/1411304.1411307.

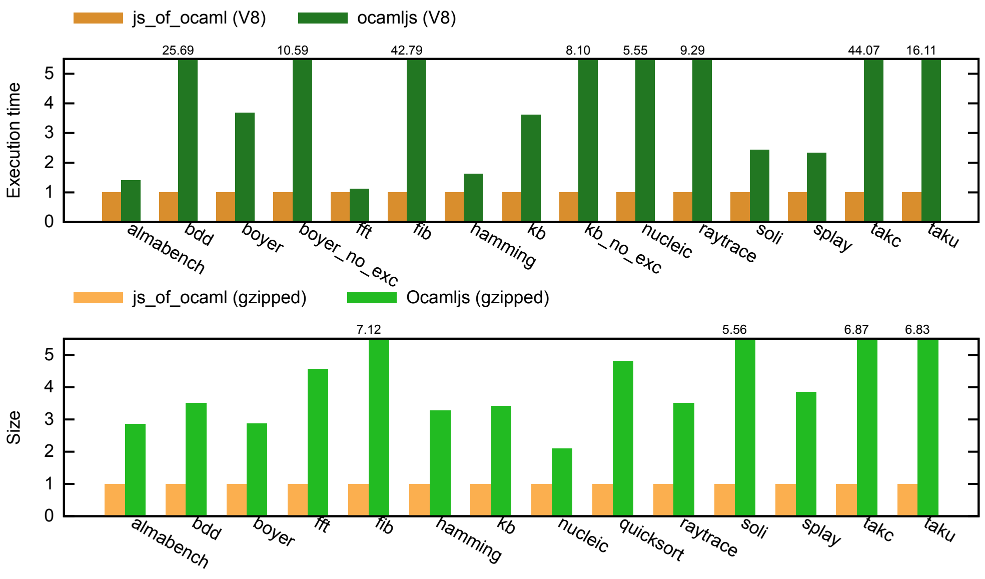

Figure 18. Comparison with Ocamljs

A. Comparison with Ocamljs

Ocamljs is a compiler from OCaml source code to Javascript, implemented as a back-end to the existing OCaml compiler [8]. Thus, contrary to Js_of_ocaml, you need to perform a distinct installation of OCaml to use Ocamljs, and you have to recompile all the libraries you may need.

Ocamljs follows a different philosophy: it attempts to merge OCaml datatypes with the corresponding Javascript datatypes. For instance, OCaml objects are implemented as Javascript objects. Conversely, Javascript objects are given an OCaml object type. A mixed representation of strings is used: mutable OCaml-style strings and immutable Javascript strings both have the same type. All this is good for interoperability, but can be a source of incompatibilities and can result in runtime errors not caught by the type checker.

Ocamljs optimizes tail recursion, but this comes at a large performance cost (See figure 18).