JFP **26**, e7, 42 pages, 2016. © Cambridge University Press 2016
doi:10.1017/S0956796816000095

OCaml-Java: The Java Virtual Machine as the target of an OCaml compiler[*]

XAVIER CLERC
(*e-mail:* xclerc@ocamljava.org)

Abstract

This article presents how the compiler from the OCaml-Java project generates Java bytecode from OCaml sources. Targeting the Java Virtual Machine (JVM) is a technological challenge, but gives access to a platform where OCaml can leverage multiple cores and access numerous libraries. We present the main design choices regarding the runtime and the various optimizations performed by the compiler that are crucial to get decent performance on a JVM. The challenge is indeed not only to generate bytecode but to generate *efficient* bytecode, and to provide a runtime library whose memory footprint does not impede the efficiency of the garbage collector. We focus on the strategies that differ from the original OCaml compiler, as the constraints are quite different on the JVM when compared to native code. The level of performance reached by the OCaml-Java compiler is assessed through benchmarks, comparing with both the original OCaml implementation and the Scala language.

1 Introduction

The OCaml-Java project is an attempt to provide seamless integration of OCaml and Java, allowing to combine both languages in a single code base. One key objective is to run OCaml code on a Java Virtual Machine (JVM), allowing to circumvent limitations of the original OCaml implementation. Even though a young project, the OCaml-Java project already provides a high level of compatibility with the original OCaml implementation, and decent performance compared with the OCaml native compiler. The OCaml-Java project has already been presented at large in Clerc (2012b), and its code generation scheme has been described broadly in Clerc (2012a). This article provides a detailed description of the compiler, as well as some key design decisions related to its runtime support.

In the remainder of this section, we provide an overview of existing compilers and justify the interest in the JVM as a target platform. In Section 2, we summarize the objectives and challenges our project faces. In Section 3, we present the compiler architecture and particularly how it differs from existing OCaml compilers. Then, in Sections 4 and 5, we respectively detail the runtime representation of values and

[*] http://www.ocamljava.org

the code generation scheme. Section 6 is devoted to the exposition of the post-compilation optimization process. Section 7 illustrates the performance level of the current implementation through various benchmarks. Finally, Sections 8 and 9 put forward related and future work.

1.1 Existing compilers

1.1.1 Original compilers

The official OCaml distribution (Leroy *et al.*, 2013) ships with two compilers, `ocamlc` and `ocamlopt`. The former compiles to a dedicated bytecode, while the latter compiles to native code. The `ocamlc` compiler, and its associated virtual machine (`ocamlrun`), are available on any 32- or 64-bit platform running any flavor of either Unix or Windows. It is noteworthy that the `ocamlrun` virtual machine does not feature a JIT compiler. It nevertheless delivers decent performance thanks to an optimized design and implementation (Leroy, 1990).

The `ocamlopt` compiler is available on fewer platforms than OCaml, but provides support for the two most widespread architectures: `amd64` and `ia32` under Linux, Mac OS X, and Windows operating systems, `powerpc` under Linux, and Mac OS X operating systems, `arm` under Linux, and `sparc` under Linux, and Solaris operating systems. The `ocamlopt` compiler produces binaries "*at worst twice as slow as binaries produced by an optimizing C compiler*" (objective stated by the project managers).

1.1.2 Alternative compilers

Besides the two compilers from the official distribution, there are two notable initiatives that propose to extend the range of targets for the OCaml compilers: `js_of_ocaml` (Vouillon & Balat, 2014), and `ocamlcc` (Mauny & Vaugon, 2012). The goal of the `js_of_ocaml` project is to support execution of OCaml programs on a JavaScript engine, thus allowing to program dynamic web pages in OCaml. This project is part of a larger endeavor, namely the Ocsigen project (Balat *et al.*, 2009), whose aim is to provide a full-stack framework for web programming, with a particular focus on using the same programming language on both the server and the client.

The goal of the `ocamlcc` project is to target the C programming language. The underlying motivation is that it will provide native performance, while still being portable. It is thus primarily useful for platforms where `ocamlopt` is not available. It may also be useful on platforms supported by `ocamlopt`, by giving access to optimizations performed by the C compiler but not performed by `ocamlopt`.

Interestingly enough, both projects are not taking OCaml sources files as their input, but rather bytecode files produced by the `ocamlc` compiler. This contrasts with our compiler that is actually an extension of the `ocamlopt` compiler. This choice, to build a toolchain on the output of `ocamlc`, is motivated by the fact that it is easier for the developer to provide an external tool rather than to plug into the compiler itself. Moreover, it is often argued that the bytecode format is more stable than the internal representation manipulated by the compiler, leading to less maintenance effort over successive OCaml versions.

1.2 The case for the Java target

Although the original compilers meet portability and performance requirements, through respectively `ocamlc` and `ocamlopt`, targeting the JVM is appealing for various reasons. The most important one is related to libraries: the number of libraries available to the OCaml developer is a known weakness of the OCaml ecosystem. Granting access to every Java library is attractive for an OCaml developer in order to reduce the time needed to develop an application. In order to allow the OCaml developer to leverage Java libraries, an extension of the OCaml type system has been designed to integrate the typing of Java elements into OCaml. The extension, presented in Clerc (2013b), supports instance creation, method calls, field accesses, and implementations of Java interfaces through OCaml code. The challenge is that the two type systems are very different: mostly versus totally static typing, nominal versus structural typing, and covariant versus invariant arrays.

Another compelling reason to target the JVM is to lift the constraint of the *global runtime lock* of the original implementation. This lock means that at any time only one thread can execute OCaml code (it is possible that several threads execute C code in parallel). While this model may prove sufficient when computations are dominated by i/o operations, it will fall short when one really needs to use the full computing power of a recent multicore CPU. One way to circumvent this limitation is to use several processes rather than several threads to perform computations in parallel. Several libraries have been developed along these lines (Danelutto & Di Cosmo, 2011; Filliâtre & Kalyanasundaram, 2011; Stolpmann, 2012), which all follow the map/reduce model. Moreover, the JoCaml dialect (Fournet *et al.*, 2003) provides language extensions for parallel and distributed programming.

While very useful, these projects do not help when one really needs to work in a shared-memory model. An option to get rid of the global runtime lock is, of course, to rewrite the OCaml runtime and make it re-entrant, and to also develop a parallel garbage collector. Making the OCaml runtime re-entrant is not very difficult, but developing a parallel garbage collector is a huge endeavor. The `ocaml4mc` project (Chailloux *et al.*, 2009) proposes such a garbage collector for the `x86_64` architecture. However, due to an extensive use of assembly language, this development would have to be re-done for each architecture to be supported.

2 Objectives and challenges

2.1 Objectives

There is a growing perception that one can look at Java under two different perspectives: as a platform, and as a language. The goal of the OCaml-Java project is to provide seamless integration with Java by leveraging both the platform and the language.

Leveraging Java as a platform is mainly the ability to run OCaml code on the JVM. This entails the following practical advantages:

- portability of compiled code;
- first-class performance level based on a JIT;
- shared-memory concurrent programming based on a parallel garbage collector.

Leveraging Java as a language is the ability to use Java libraries from OCaml sources. To this end, it is not enough to provide a compiler producing Java bytecode, it is also necessary to provide means to manipulate Java entities inside OCaml applications. For example, the following code shows how to read a mp3 file:

```
let read_mp3 url =
  let media = Java.make "Media(_)" url in
  let player = Java.make "MediaPlayer(_)" media in
  Java.call "MediaPlayer.setOnEndOfMedia(_)" player
    (Java.proxy "Runnable" (object
        method run = exit 0
    end));
  Java.call "MediaPlayer.play()" player.
```

The `Java.make` function allows invoking constructors, while the `Java.call` function allows invoking methods. The `Java.proxy` method allows to implement a Java interface (`Runnable` in the example above) with OCaml code.

Handling of Java elements is done through an extension of the type system that supports the creation and manipulation of Java objects from the OCaml language, with no costly indirection. The extension to the type system is presented in Clerc (2013b); two important properties of this extension are that (i) OCaml syntax is unchanged (thus meaning that all tools working at the source level can still be used), and (ii) Java manipulations are directly translated to plain bytecode (thus meaning that there is no reflection involved and no additional runtime cost incurred).

The extension of the type system is a layer on top of the original type system. The layer encodes the type of Java instances into bare OCaml types, using a combination of phantom types and polymorphic variants. The layer is also able to decode such OCaml types in order to print error messages with easy-to-read references to Java classes. The typing of Java elements is thus embedded into the OCaml type system, unifying both worlds.

Of course, the goal of providing seamless integration also means that the OCaml-Java project provides ways to access OCaml from Java. The ability to call OCaml code from Java is useful, for example, to develop plugins for Java applications. This can be done mainly by using the following:

- the Java scripting framework that supports the evaluation of code snippets written using foreign languages;

- the `ocamlwrap` (see Section 8.1) tool (Clerc, 2013a) that supports the generation of Java class definitions to easily manipulate OCaml values and functions.

2.2 Challenges

The challenges we face are indeed mainly the consequence of the obvious and implicit goal to provide source compatibility with the original OCaml implementation, and to present a compiler with decent performance. For example, OCaml uses *tagged* integers to encode some simple values but Java does not provide an equivalent mechanism. Likewise, OCaml uses exceptions for control flow manipulation while Java uses them only for error handling. Still in the performance department, the original OCaml compilers implement tail call optimization, while Java does not provide out-of-the-box support for them. Finally, OCaml supports separate compilation and handles polymorphism by having a single runtime type for all values.

These challenges are quite different from those encountered by other functional languages targeting the JVM, such as Scala (Odersky *et al.*, 2003) or Clojure (Hickey, 2008). Indeed, these language have been designed from the ground up to run on a JVM. This means that they have the possibility to rule out features that would be too costly in terms of either CPU or memory usage; for example, we are tied to the value representation of the existing compilers in order to remain compatible with legacy code. The situation of F# (Syme *et al.*, 2005) on the .NET platform is similar in the sense that the object-oriented part of the language was designed to be compatible with the object model of C# (Microsoft, 2000).

In this respect, the work presented here is closer to projects like OCamIL (Montelatici *et al.*, 2005), MLj (Benton *et al.*, 1998; Benton & Kennedy, 1999), or SML.net (Benton *et al.*, 2004) that have to support an existing language and accommodate the limitations of the target platform. Fortunately, thanks to recent evolutions of the JVM, it is now easier to get decent performance for a functional language. Most notably, Java 1.7 introduced the *G1* garbage collector, and method handles.

The *G1* garbage collector is more suited to functional languages than the previous implementation, as its performance will not plummet under heavy allocation of short-lived objects. It means that the allocation/collection pattern observed in most functional programs will no longer deteriorate performance in dramatic proportions. Nevertheless, we still need to be careful when designing our compilation schema and data representation, in order to avoid unnecessary boxing.

Method handles, that are akin to function pointers in C, provide a simple and efficient way to encode closures. Before method handles, there were basically two techniques to implement closures: generation of multiples classes (one per method to be called) implementing a given interface, or use of reflection to dynamically call a given method. The former was a burden at both compilation time, generating many almost-identical classes, and at runtime, polluting the classloader with the generated classes. The latter was a burden at runtime because relying on reflection implies to box all values, and is very costly performance-wise.

Table 1. *Files manipulated by the various compilers*

Compiler	ocamlc	ocamlopt	ocamljava
Interface source	.mli	.mli	.mli
Implementation source	.ml	.ml	.ml
Compiled interface	.cmi	.cmi	.cmi
Compiled implementation	.cmo	.cmx	.cmj
Compiled library	.cma	.cmxa	.cmja
Compiled plugin*	.cmo	.cmxs	.cmjs
Object binary	...	.o / .obj	.jo
Library binary	...	.a, .so, ...	.ja

* A *plugin* is a module that can be dynamically loaded.

3 Compiler architecture

In this section, we will start by presenting how the original OCaml compilers transform a source file into, respectively, bytecode and native code. Then, we will explain how the same transformation occurs in the newly introduced ocamljava compiler, underlining the differences with the original compilers.

Table 1 shows the extensions used for the different file kinds manipulated by the compilers. All compilers share, of course, the same files for input sources; they also produce the identical files for compiled interfaces. All compilers rely on the principle of separate compilation. Through compiler execution, every .mli/.ml couple defining an OCaml top-level module is first transformed into several files containing:

- typing information about exported elements (.cmi file);
- list of imported modules with digests to ensure that all modules are compiled against the same version (.cmi, .cmo, .cmx, and .cmj files);
- inlining information, in order to to perform cross-module inlining (.cmx, and .cmj files).

The code for a module is stored by ocamlc inside .cmo files, while ocamlopt and ocamljava store code inside dedicated files using, respectively, .o and .jo files. Such .o files can be directly passed to the linker of the underlying platform. Once each module has been compiled, implementation files can be linked together to produce either a library or a standalone executable. The ocamljava compiler uses the .ja extension for library binaries to differentiate them from executable .jar files. While both .jo and .ja files are indeed Java archives, they do not use the .jar extension to distinguish them from Java-generated files.

3.1 Original compilers

Both ocamlc and ocamlopt compilers naturally share a large code base: parsing and typing are identical, using the very same code. Figure 1 shows the successive

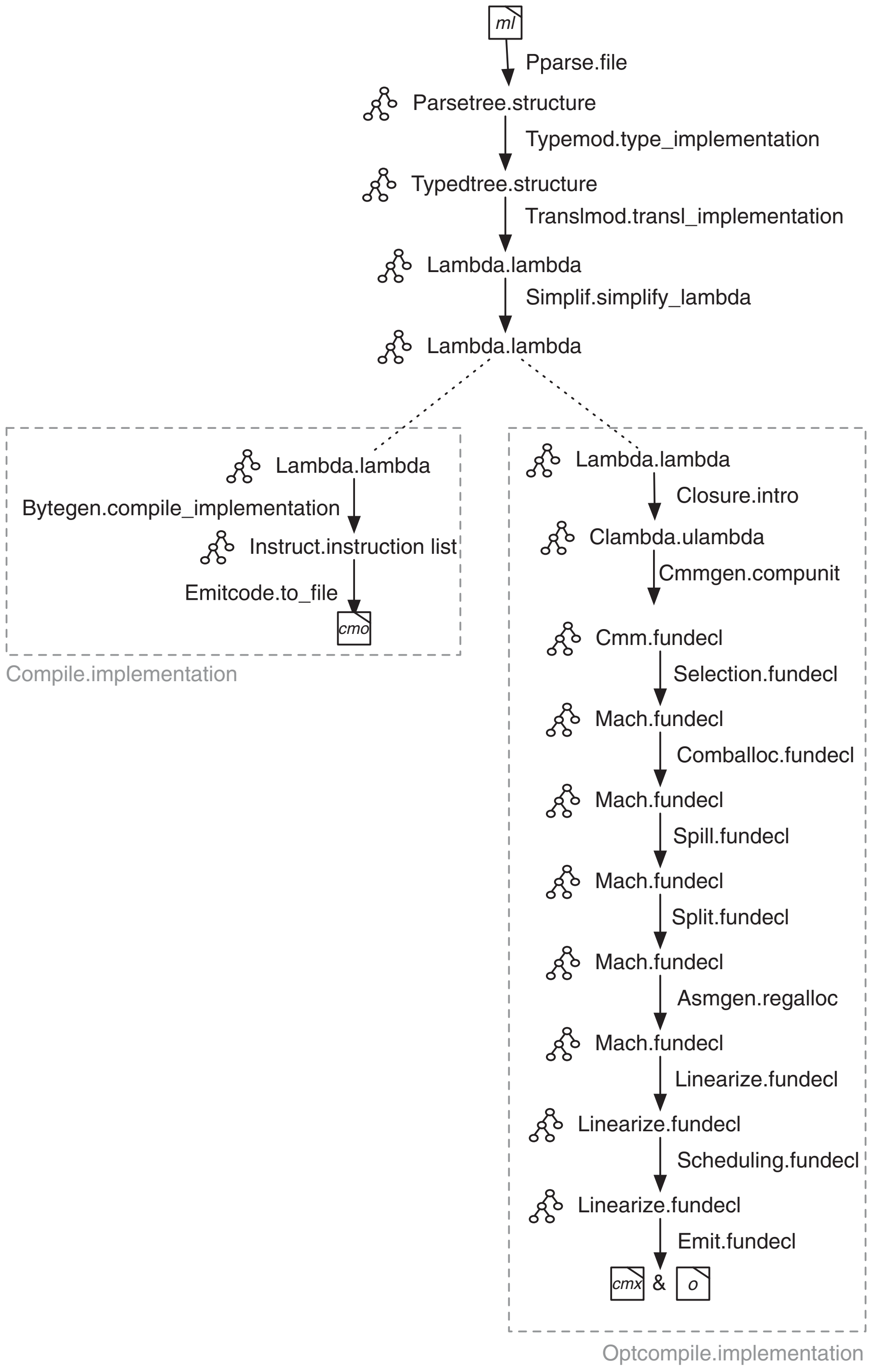

Fig. 1. Passes of original OCaml compilers.

passes of both compilers from an implementation source file to an implementation compiled file. We do not detail the compilation of an interface source file because it (i) does not produce code, and (ii) is identical in both compilers.

Figure 1 presents the various passes from a source file to a binary file, as well as the different data structures used during the process. We only skip the passes that are just intended to optionally pretty-print the intermediate data structures for debugging. As previously stated, both compilers share the passes related to parsing (`Pparse.file`) and typing (`Typemod.type_implementation`). They also share the very first passes related to code generation: `Translmod.transl_implementation` and `Simplif.simplify_lambda`. These passes produce so-called *lambda code*, which is the most abstract representation of code to be compiled. One distinctive characteristic of lambda code is that it is untyped. Indeed, the compilation process is based on type erasure, and relies on the strong and static typing of OCaml to ensure that no type check is ever needed at runtime. It is also noteworthy that constructs such as functors and first-class modules have been converted to, respectively, functions and bare data blocks, which means that no special treatment is required to compile them.

After lambda code, the two compilers diverge. The bytecode compiler only needs two more passes to produce its result; these passes are straightforward because the instruction set of the OCaml virtual machine is essentially based on the model of the lambda code. Of course, the native compiler has far more work to perform because it has to accommodate an instruction set that was not specifically designed for functional programming, and has to target a register-based machine rather than a stack-based machine.

The first `ocamlopt`-specific step, `Closure.intro`, handles the transformations associated with closures (e.g. introducing explicit environment parameters where needed), and related optimizations. These optimizations include inlining, basic constant propagation, and identification of *direct* calls. A function call is direct if the function is statically known, as opposed for example to a function passed as a parameter or retrieved from a data structure. This optimization is crucial, because direct calls often account for the vast majority of calls, and are dramatically cheaper than generic function applications.

The `Closure.intro` function produces a variant of lambda code named *C lambda*, the latter differing from the former mainly concerning function calls. Then `Cmmgen.compunit` transforms *C lambda* into *C*-- (i.e. *C minus minus*), which is lower-level and contains for example information about data size and alignment. The `Selection.fundecl` function is responsible for producing *machine code*, which is an abstract representation that is still largely independent from the target platform, based on pseudo-instructions. The `Comballoc.fundecl` function optimizes memory allocations performed in the same code block in order to merge them when possible.

The next three phases of the native compiler, namely `Spill.fundecl`, `Split.fundecl`, and `Asmgen.regalloc` implement register allocation. This first determines the liveness of manipulated elements before actual register allocation is done using a graph-coloring algorithm. The algorithm is the same for every supported platform, but is of course fed with platform-specific information such as the number of available registers.

Finally, `Linearize.fundecl` reifies pseudo-instructions into lists of instructions for the target architecture, and `Scheduling.fundecl` optimizes the resulting order. The very last step is to turn these instructions into a proper binary file. To this end, the `ocamlopt` compiler outputs the assembly source for the instructions to a temporary file, and executes an external assembler to produce the platform-specific `.o` file.

3.2 OCaml-Java compiler

The OCaml-Java compiler follows the design of the `ocamlc` and `ocamlopt` compilers. In this respect, it is very different from the `js_of_ocaml` and `ocamlcc` projects whose inputs are the bytecode files output by the `ocamlc` compiler. There are three major reasons to justify this design choice:

- Java puts a hard (and low) limit on the maximum size of a Java method (64 kb), meaning that it is impractical to have to treat the program as a single whole (`ocamlc`-produced bytecode is monolithic);
- the OCaml-Java compiler does not only produce Java bytecode, it also provides extensions to the type system in order to manipulate Java instances from an OCaml program;
- implementing the unboxing strategy is easier, because we only have to propagate types through compiler phases, while projects based on bytecode have to reconstruct (an approximation of) this information.

The OCaml-Java compiler can be seen as a third branch of the tree depicted by Figure 1. This means that passes up to `Simplif.simplify_lambda` are shared with the original compilers. Only a small modification is made to the lambda code structure, in order to keep type information related to function declarations and calls (see 5.4). This type information is used in subsequent passes to perform some optimizations, such as unboxing. Sharing passes with the original compilers keeps maintenance costs low. The most important part of the cost in indeed related to the update of patches responsible for type propagation, which is scattered across less than ten files.

Figure 2 shows which transformations are then performed on lambda code. First, very similarly to the native compiler, `Jclosure.jlambda_of_lambda` is responsible for the handling of closures, producing a slightly optimized *Jlambda*. Just like `ocamlopt`, this pass is responsible for the identification of direct calls. It also performs some other optimizations, such as loop unrolling and use of *static* exceptions (these optimizations are discussed in Section 5).

Then, `Macrogen.translate` decomposes operations from the *Jlambda code* into *macro instructions* that are not Java bytecode instructions but can be easily mapped to them. This pass is also responsible for variable allocation which entails the choice of their actual representation, and hence value unboxing. Here, the typing

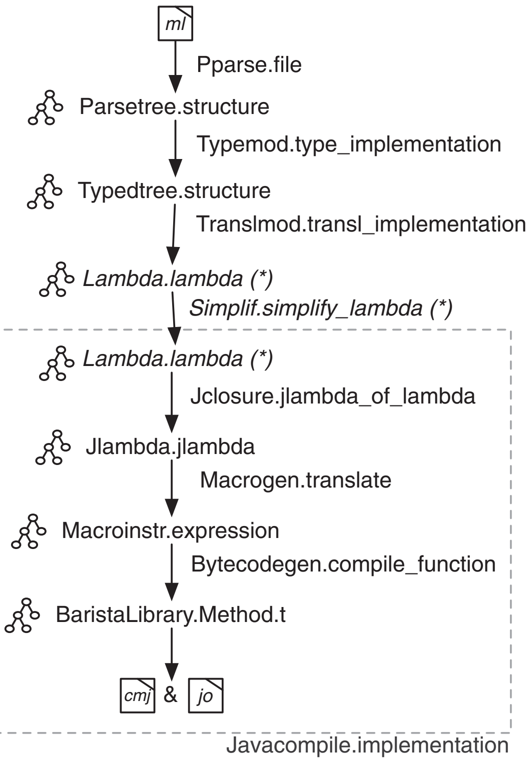

Fig. 2. Architecture of OCaml-Java compiler.

information added to *lambda code* plays a key role, allowing to determine optimal data representation with no effort.

Finally, the last compiler pass, `Bytecodegen.compile_function`, produces actual Java bytecode using the Barista library to build an in-memory representation of the class file to emit. This pass is quite straightforward as boilerplate operations such as the computation of stack maps are handled by the Barista library. Indeed, the only important optimization handled by this pass is tail-call optimization (see 5.1).

The point where `ocamlopt` and `ocamljava` compilers diverge (that is the function named `Jclosure.jlambda_of_lambda`) has been chosen because the latter has to be more aggressive regarding constants handling and propagation. Indeed, as presented in Section 4, the native compiler uses a uniform representation and does not need to optimize `int` values, as they are always unboxed (using a tagged representation). On the other hand, our compiler stores `int` values as boxed values (in order to keep a uniform representation despite the Java distinction between primitive and object values), but tries to unbox values as much as possible when performing computations in functions.

Another construct is treated in a different way by the `ocamljava` compiler: switches, as the Java instruction set features both table and lookup instructions while the native code generator only emits code corresponding to table switches. Lookup switches are useful because they can be used where the original OCaml compiler would emit intertwined jumps. The original compiler emits such jumps when the different values to test are not contiguous. For example, in the following program:

```
let f = function
| 0 -> ...
| 1 -> ...
| 2 -> ...
| 3 -> ...
| _ -> ...
let g = function
| 0 -> ...
| 100 -> ...
| 200 -> ...
| 300 -> ...
| _ -> ...
```

`ocamlopt` will compile `f` to a switch (defining an array of destinations and jumping to the destination at the passed index/value), but produce the following pseudo-code for `g`:

```
if (param >= 101) then
  if (param != 200) then
    if (param != 300)
      ...
    else
      ...
  else
    ...
else
  if (param != 0) then
    if (param >= 100) then
      ...
    else
      ...
  else
    ...
```

`ocamljava` will compile `f` to a `tableswitch`, i.e. an index-based jump:

```
label_f: push param
         tableswitch default: 0 3
                     label_0:
                     label_1:
                     label_2:
                     label_3:
label_0: ...
```

```
label_1: ...
label_2: ...
label_3: ...
default: ...
```

and will compile g to a `lookupswitch`, i.e. defined by key/destination couples:

```
label_g:   push param
           lookupswitch default:
                        0   => label_0
                        100 => label_100
                        200 => label_200
                        300 => label_300
label_0:   ...
label_100: ...
label_200: ...
label_300: ...
default:   ...
```

In both cases, `ocamljava` produces the same code that would be produced by a Java compiler.

When compiling a top-level module named `Module` from a file named `module.ml`, two files are produced: a `module.cmj` file corresponding to the `.cmx` file of the native compiler, and a `module.jo` file corresponding to its `.o` file. The `module.jo` file is actually a Java archive containing three entries:

- `Module.class` is the class file containing the implementation of all module functions as Java static methods, as well as an `entry()` method executing the module initialization code;
- `Module$Global.class` is the class file used to hold all global variables for the module (such variables can be mutable);
- `Module$Constants.class` is the class file used to hold all structured constants for the module (such values, for example strings or arrays can be mutated).

The class `Module.class` can be safely shared by several programs in the very same JVM, as it holds no state. The `Module$Global.class` and `Module $Constants.class` classes hold the state, and there should be one instance per running program.

Compiled modules are later linked to produce an executable `.jar` file. The `ocamljava` compiler currently supports four linking modes:

- *genuine applications*, that are bundled as standalone executable `.jar` files;
- *applets*, that are designed to be run inside a web browser to provide interactivity to web pages;
- *servlets*, that are designed to be run in containers serving web pages computed on the server side;
- *scripts*, that are used to provide support for the Java scripting framework (implemented by the `javax.script` package).

The linking of modules involves the creation of the entry point of the program (e.g. a class named `ocamljavaMain` in the case of a *genuine application*), that is responsible for calling the `entry()` method of each linked module, in the appropriate order. This also involves the loading of modules' constants and initialization of global variables. Both constants and globals of a module are stored using thread-local variables.

The use of thread-local variables introduces an arguably unnecessary indirection, but is actually needed as several OCaml programs may run inside the very same JVM (e.g. through several applets on the same page, or several servlets in the same container), with each program keeping its own copy of the data. This is even needed for constants as, despite their misleading name, some of them are indeed mutable (their name stems from the fact that they appear as constant literals in the source file). For example, OCaml string is actually mutable byte arrays.

The key points are that (i) threads are managed by the OCaml-Java runtime, and that (ii) a given thread is attached to a given program. It is thus fine to access the data of a program through thread-local variables, allowing to share the very same code for several programs (e.g. the code of the standard library). The `entry()` method of a module is responsible for initializing the module data of a given program.

This indirection incurs a runtime overhead that should ideally be paid only when one is running several programs inside the very same JVM. As a consequence, a tool named `ocamljar`, whose detailed role is presented in Section 6, has been developed to optimize `.jar` files produced by the compiler. The motivation for such a tool is that it is easier to only have one version of the compiled modules and then to only modify the resulting `.jar` rather than to deal with multiple module versions compiled with use of thread-local storage turned either on or off. Moreover, the `ocamljar` tool, working on complete programs, it able to perform optimizations permitted only when whole-program information is available.

4 Value representation

The compilation scheme of OCaml performs type erasure, meaning that *almost* all typing information is lost during the compilation process. This is of course not a problem as OCaml is statically and strongly typed, meaning that no type test has to be performed at runtime.

Basically, all values share a common representation, namely `value` in the original runtime, written in C. Having a common type for all values at runtime greatly simplifies the compilation process because such a common representation makes polymorphism compilation easier, particularly in the case of separate compilation.

More precisely, the use of the `value` type is mandatory at function boundaries (i.e. to call an OCaml function, or a C primitive), but a function is free to use whatever representation it prefers for local values. This freedom is indeed crucial in order to reach good performance because it supports unboxing of values. Values still need to be boxed at function's call site, but this penalty can also be partially avoided through function inlining.

In the remainder of this section, we first present the *de facto* specification of runtime values set by the original OCaml implementation, and then present how

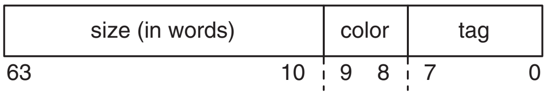

Fig. 3. Header of an OCaml block (on a 64-bit architecture).

such a specification is implemented in OCaml-Java. It is noteworthy that OCaml-Java has to closely follow the value representation set by the original runtime, because some OCaml core libraries rely on the value representation. For example, the implementation of both `Printf` and `Scanf` modules relies on the precise memory layout of closures.

For example, in order to handle the expression `Printf.printf "%d -> %s" 1 "one"`, the compiler:

- parses the format string to determine the number and types of arguments;
- ensures that passed arguments are actually compatible;
- generates a call to the `Printf.printf` function with one argument that is the format string, this call returning a closure;
- generates an application of this closures with parameters `1` and `"one"`.

At runtime, the `Printf.printf` function parses the passed (format) string in order to determine the number and types of arguments, and uses this information to determine the shape of the closure. Then, when actually printing the elements on the standard output, the function accesses the closure parameters.

4.1 Original runtime

The various values manipulated at runtime by an OCaml program can be specified by the following grammar:

value	::=	long unboxed value
	|	pointer to managed block
	|	pointer to unmanaged block.

A long value (used for example to encode the OCaml `int` type) is differentiated from a pointer value using tagging: the lowest bit is set to one for long values, while it is set to zero for pointer values (which is fine as memory addresses are always even). The encoding of an integer value i as a long unboxed value l is thus done according to the following equation: $l = (i \ll 2) + 1$. A managed pointer (i.e. inside the OCaml heap) is discriminated from an unmanaged one (i.e. allocated by C code) by keeping the list of memory blocks allocated as parts of the OCaml heap. To this end, the current implementation uses an hash table that is used for example by the garbage collector to know whether it should follow a given pointer.

A managed block is made of two parts: its header and its contents. The header, depicted by Figure 3 contains three elements: (i) the total size of the block, (ii) its color (used by the garbage collector), and (iii) its tag. The tag is used to know how the contents of the block should be interpreted. The following grammar offers a simplified view of the various possible block kinds:

managed block ::= header with tag ⊕ array of *size* values
| header with *string* tag ⊕ array of bytes, padded to *size*
| header with *double* tag ⊕ 64-bit float value
| header with *double array* tag ⊕ array of *size* 64-bit float values
| header with *closure* tag ⊕ code pointer ⊕ array of *size - 1* values
| header with *object* tag ⊕ pointer to method table ⊕object identifier ⊕ array of *size - 2* values (instance variables)
| header with *lazy* tag ⊕ thunk value
| header with *forward* tag ⊕ value
| header with *abstract* tag ⊕ pointer to C heap
| header with *custom* tag ⊕ pointer to custom operations ⊕ raw data.

In the first case, the tag is not used to denote the kind of element(s) stored by the block, but is used to discriminate between the various cases of a sum type. For example, the following values `x` and `y`:

```
type t1 = A of int | B of string
let x = A 3
let y = B "abcd"
```

would be respectively represented by the following:

- a block with tag 0, and a nested integer value of `7` (tagged integer);
- a block with tag 1, and a nested block with *string* tag holding an array of 4 bytes.

Given the possible contents of a managed block, we see that some typing information is retained at runtime. However, this is not enough to recover the typing information present in the source, because several different types in the source are mapped to the same runtime representation. For example:

- `0`, `false`, `[]`, and `A` (under e.g. the type definition `A | B`) are all encoded by the long value zero;
- similarly, `::` (list constructor), `(x, y)` (couple), and `{ a; b }` (record with two elements) are all encoded using a block with tag zero and two nested values.

Again, strong typing has been enforced at compile time, so no confusion could be made at runtime between values of different types.

Most tags are self-explanatory, but *lazy*, *forward*, *abstract*, and *custom* tags deserve an additional explanation. *Lazy* tags are, obviously, used to represent a non-evaluated lazy value whose associated code is stored as a nested value. For example, `lazy expr` is compiled to a block with a `lazy` tag holding a pointer to a function `fun () -> expr`. When the value is forced, the function is called, and the following modifications occur:

- the tag is changed from `lazy` to *forward*;
- the nested value is changed from `fun () -> expr` to the result of the function evaluation.

The *abstract* tag is used to wrap a pointer to the C heap into an OCaml value. As we have seen before, it is also possible to directly use a C pointer as a value, but wrapping it as an OCaml value is encouraged as it provides a more uniform representation of values, and also alleviates the work done by the garbage collector to know whether a pointer should be followed.

Finally, the *custom* tag is used for types that optionally come with specific primitives for comparison, hashing, and (de-)serialization. It is noteworthy that OCaml `int32`, `int64`, and `nativeint` types are all encoded by custom blocks, meaning that besides the `int` type, all OCaml integer types are boxed. This implies that it is crucial to devise an appropriate unboxing strategy to be able to get decent performance for code based on these types.

4.2 OCaml-Java runtime

The representation of values is based on multiple classes for the various kinds of values. All classes inherit from a parent `Value` abstract class. This class implements the operations for all the kinds of values, possibly proposing a dummy or failing implementation. It is then the responsibility of children classes to override that base implementation with a correct one. The guarantee that a dummy or failing implementation will never be called is based on the static and strong typing occurring at compile time. We opted for a class rather than an interface in order to have more control over derived classes: for instance, forbidding *third-party* implementations by having only package-level constructors.

Derived classes are defined for long values, string values, double values, double array values, and block values. Contrary to the original runtime, all values even long ones are allocated because the JVM does not support tagged values. However, every creation of a value has to be done through a factory method, which allows us to share values through a cache. As an example, long values are immutable and a cache shares values between −128 and 255. These values are allocated once, at program startup, and then reference comparisons are used for values between the bounds. Caching is important from a performance standpoint: for example testing whether a list is empty by a simple reference comparison. In order to be fully compatible with the original implementation, we use 63-bit arithmetic. Indeed, the original implementation uses 1-bit tagging to tell integers and pointers apart.

The compilation scheme of OCaml turns a type such as a record or a tuple of values into a mere block at runtime. Again, strong and static typing ensures that the program will not try to access an element that does not exist (e.g. trying to access the third component of a pair). For this reason, the original OCaml compilers do not generate code for testing such bounds. However, in Java it is not possible to remove bound checks when accessing the elements of an array.[1] As a consequence, if the elements of a block were stored into an array, we would have to pay the price of a bound check at every access. Moreover, due to the covariant nature of Java arrays, each array store operation incurs a check that the actual class of the object to be stored is correct with respect to the array type.

For this very reason, we resort to what could be called *data unrolling*. Rather than having only one class named `BasicBlockValue` storing its elements as one `Value[]` field, we define several classes named `BasicBlockValue`*n* that store *n* elements as *n* `Value` fields. This allows us to define methods such as `get0()` that returns the first element of a value with no bound check.

Experimentation showed measurable speedups when growing the *n* value up to 8. The current version of the runtime hence contains classes with *n* ranging from 0 to 8. The source code for these classes is, of course, generated. Also, in addition to those classes, a `BasicBlockValue` is defined to be able to store an unbounded number of elements in an array. Then, array bound checks cannot be avoided but experience indicates that this representation is indeed used for OCaml types that turn out to be arrays, and should indeed test bounds at runtime for every access.

Figure 4 shows a slightly simplified version of the class hierarchy used to encode OCaml values when running a program on the JVM. As previously stated, container classes (used for blocks) come in several versions according to their sizes. This encoding is not only used for bare blocks (class `BasicBlockValue`) but also for blocks holding double arrays (class `DoubleArrayBlockValue`), and blocks holding long arrays (class `LongBlockValue`). Long blocks do not enjoy a specific representation in the original OCaml runtime, but `ocamlopt` uses the fact that an array contains only long values to avoid write barriers when working with such blocks. In OCaml-Java, a special representation is used, alleviating the need to box long values when they are stored inside the block.

The class named `MethodHandleValue` is used to represent closures, storing arguments from partial application into a simple block. An instance of `MethodHandleValue` holds, as its name suggests, a handle to the static method implementing the function. Instances of `MethodHandleValue` are used only when a function is either passed to a higher order function, or when partial application is used. Indeed, when a totally applied call is made to a function that is known at compile time, it results in a single `INVOKESTATIC` instruction. Lambda expressions are compiled as classical functions, after being given fresh names.

[1] The HotSpot compiler can remove such tests if it can *prove* that no illegal access will happen, but the developer cannot request to remove such tests. Albeit, as we will see in Section 6, it is possible to use *unsafe* operations akin to pointer arithmetic in Java.

```
Value
∟ LongValue
∟ MethodHandleValue
∟ BlockValue
  ∟ DoubleValue
  ∟ AbstractCustomValue
    ∟ Int32Value
    ∟ Int64Value
    ∟ NativeIntValue
  ∟ StringValue
  ∟ CachedStringValue
  ∟ AbstractBasicBlockValue
    ∟ BasicBlockValue0, ..., BasicBlockValue8
    ∟ BasicBlockValue
    ∟ LargeBasicBlockValue
  ∟ AbstractLongBlockValue
    ∟ LongBlockValue0, ..., LongBlockValue8
    ∟ LongBlockValue
    ∟ LargeLongBlockValue
  ∟ AbstractDoubleArrayValue
    ∟ DoubleArrayValue0, ..., DoubleArrayValue8
    ∟ DoubleArrayValue
    ∟ LargeDoubleArrayValue
```

Fig. 4. Hierarchy of classes representing OCaml values.

The compilation of a function capturing values is done by building and environment that is basically a tuple of the captured values. This environment is then passed to the function as an additional parameter. For example, the following function:

```
let f x y =
  x + y + z + t
```

is actually compiled as:

```
let f x y (env_z, env_t) =
  x + y + env_z + env_t.
```

It is also noteworthy that functors are compiled as simple functions. For example, the following functor:

```
module Functor (S : Signature) = struct
  let some_value = ...
  let some_function x y = ..
  let some_other_function x y z = ...
end
```

will be basically compiled as a function returning a record:

```
let functor s =
  { some_value = ...;
    some_function = fun x y -> ...;
    some_other_function = fun x y z -> ...; }.
```

The `StringValue` and `CachedStringValue` classes are used to implement string values, that are mutable in OCaml while the Java `String` class is immutable. Both `StringValue` and `CachedStringValue` classes thus implement strings as byte arrays, and provide conversion methods. A `CachedStringValue` instance also contains an actual `String` instance, that is the *cached* conversion of the byte array, in order to avoid repeated conversions.

The *large* blocks in Figure 4 are developed to handle the storage of blocks that contain more values than a Java array can store (a Java array uses 32-bit indexes, while an OCaml array uses 63-bit indexes). To this end, they use two-dimensional arrays, thus allowing to store values indexed by a Java `long` rather than a Java `int`.

4.3 Alternative encodings

At first, one may wonder why the encoding of values in OCaml-Java is a direct translation of the encoding set by the original compilers. The use of tags, in particular, seems superfluous as different Java classes can be used to discriminate between the various kinds of blocks. Unfortunately, we are constrained by the need to be compatible with core libraries of the OCaml distribution that have implementations based on the low-level memory layout of values. As an example, the `Printf` and `Scanf` modules directly manipulate closures, thus enforcing to use the a memory layout akin to the original one, at the expense of lost optimization opportunities. Of course, such libraries could be rewritten, but it would increase the maintenance cost of the project, and also leave unanswered the question of third-party libraries (that may, as well, be based on such low-level implementation details).

Even under those constraints, other encoding schemes could be devised, and previous versions explored some alternatives. We experimented with an encoding based on the classes from `java.lang` with `Object` rather than `Value` as the parent class of all values, but performance was inferior due to the number of casts to perform. Another scheme was used in versions 1.*x* of the project: rather than having multiple subclasses, only one `Value` class was used for every kinds of values. In order to avoid casts, we used multiple fields to store the multiple kinds of values. This encoding led not only to a waste of memory, but also to a great performance penalty as the garbage collector had far more references to iterate over (most of them being null values).

When comparing the encoding scheme to the ones of other JVM languages, it is important to only compare to languages sharing the same constraints: whether the implementation has to be compatible with an existing reference implementation, or is for a brand new language. Indeed, languages such as Clojure (Hickey, 2008) or Scala (Odersky *et al.*, 2003) are completely free to design their encoding schemes because they do not have to abide by an existing specification. At the opposite, projects such as JRuby (Nutter *et al.*, 2008) or OCaml-Java have a more constrained design space. For example, the idea of *data unrolling* in order to avoid array bound checks is also used in JRuby.

4.4 Constants and globals

Constants are used only in the compiled module and are grouped together only for practical purpose, meaning that they are never used as a whole. Conversely, globals can be used by foreign modules and are used as a whole as the module representation (needed for example for functor application or use as first-class module). Constants and globals also differ in the sense that constant values are by definition always used to designate boxed (i.e. allocated) values while global values can be either boxed or unboxed. These differences lead to different storage policies for constants and globals.

The constant values for a module are stored in instances of a `Module$Constants` class that only contains a field of type `Value` for each constant. The class inherits from `java.lang.Object` and does not provide any method; it is only used as a data container.

The global values for a module `Module` are stored in instances of a `Module$Global` class that contains a field for each value. The type of the field depends on the type of the stored value, meaning that an OCaml `int64` value will be stored in a Java `long` field. The class inherits from `Value` as it will be used as a classical OCaml value by functors. Inheriting from `Value` also means that the class has to provide accessors to its fields through `get`/`set` methods, so that it can be used as any other boxed value. However, when some code accesses to a given field, it can also directly read or write the corresponding field, short-circuiting the accessor method. This is particularly important when the value is stored unboxed as the use of an accessor method would result in boxing/unboxing.

5 Bytecode generation

In this section, we detail how bytecode is actually generated by the `ocamljava` compiler, and highlight the key points that make the compiler different from `ocamlopt`. We first present the overall generation scheme and its use of the Barista library. Then, we explain how exceptions and primitives are handled. Finally, we discuss how type propagation is used by the compiler, and how loops are optimized.

5.1 General scheme

As already seen in Section 3, compilation from lambda code to Java bytecode is done in three steps:

1. `Jclosure.jlambda_of_lambda` handles the transformation of closures, and identifies *direct* calls;
2. `Macrogen.translate` handles the choice of data representation;
3. `Bytecodegen.compile_function` produces an in-memory representation of the Java class file.

The `Jclosure.jlambda_of_lambda` function determines whether a function call can be translated into a mere Java static method invocation, or has to go through the more general mechanism of function application (that can be partial or total). It is noteworthy that tail-call optimization is not performed at this step (see below).

The `Macrogen.translate` function determines how values are locally stored by compiled functions. Most notably, this implies to choose between boxed and unboxed representations for integer and float types. This is a crucial operation as we observed a gain in the 25%–33% interval between programs without any unboxing and the current strategy.

The `Bytecodegen.compile_function` function is responsible for a wide range of low-level optimizations that are specific to OCaml-Java, such as the use of specialized block containers (as seen at Section 4) or the use of the value cache (holding integers constants, just like the JVM does for `Integer` instances). It produces the code for each function to be compiled as a sequence of bytecode instructions. The `Bytecodegen.compile_function` function also optimizes direct calls to the current function, thus performing (a restricted) tail-call optimization. This means that the tail-calls optimized by the `ocamlopt` but not optimized by `ocamljava` are the tail-calls between mutually recursive functions. In practice, supporting optimization of tail-calls only for directly recursive functions is enough for most programs.

The resulting sequence of bytecode instructions is then passed to the underlying Barista library (Clerc, 2007) that is in charge of boiler-plate operations over bytecode, with no knowledge of the OCaml-Java specifics. The Barista library first builds a control-flow graph from the sequence of instructions, and then uses that graph as the basis for various optimizations. The nodes of the graph are labeled with instruction sequences, and the graph structure encodes the control flow of the function. The optimizations can be split into two categories:

- non-structural optimizations that are modifications to the instruction sequence attached to a node;
- structural optimizations that are modifications to the graph structure.

Non-structural optimizations are simple peephole optimizations that modify the instruction sequence for better size and speed. Better size is obtained by changing generic instruction into specialized one (e.g. using `ALOAD_0` instead of `ALOAD 0`). Better speed is obtained by removing unnecessary operations (e.g. pushing a static field value and then popping that value), and taking advantage of neutral/absorbing elements as well as removing identity operations.

Structural optimizations of the graph include optimization of jumps over jumps (thus short-circuiting empty nodes), removal of nodes that cannot be reached (thus removing dead code), and partial evaluation. Partial evaluation supports control-flow-aware constant folding, and optimization of conditional jumps whose condition is statically known into unconditional jumps.

Finally, the Barista library uses the graph to compute the stack information to be recorded in the Java file along with each method:

- the maximum size of locals;
- the maximum size of the operand stack;
- the stack frame at each branching point.

The maximum sizes have always been used by the JVM for safety reasons, while the stack frames are used since Java 1.6 in order to speed up the class file verification. By providing explicitly the stack frame, the JVM verifier only has to check that the frame is consistent with the instructions, while in previous versions it had to actually infer the frame, which is more costly. Providing stack frames thus reduces the startup time of a Java application, and is mandatory in recent Java versions.

5.2 Handling of exceptions

Both OCaml and Java languages provide support for exceptions, and it is thus natural to translate OCaml exceptions to Java ones. However, a problem arises as the semantics differ:

- in OCaml, when an exception is raised, only the stack elements that have been pushed since the begin of the protected block has been entered are popped;
- in Java, when an exception is raised, the operand stack is emptied.

Of course, the semantics of Java is perfectly legitimate as a `try/catch` construct is a statement and cannot thus appear in an expression. Practically, this means that although `try/catch` statements can be nested, the operand stack is always empty at the beginning of each statement. On the opposite, the `try/with` construct of OCaml is an expression that can then appear inside another expression, as in the following example:

```
let x =
  funct
    arg0
    (try ... arg1 ... with _ -> ... arg1' ...)
    ...
    argn.
```

As a consequence, a naive mapping of an OCaml exception to a Java exception would discard all value on the stack whenever an exception is caught. In the previous example, the JVM would discard the `arg0` value. The naive bytecode would be akin to:

```
      push arg0
try:  push arg1
      goto next
with: push arg1'
next: push arg2
      ...
      push argn
      invoke funct
```

with an exception table stating that any exception occurring between labels `try` and `with` should transfer control to `with`. However, this bytecode would not pass verification. There are two paths to the `next` label: with and with no exception.

With no exception, the operand stack contains `arg0` and `arg1`; with exception, the operand stack contains only `arg1'`. The JVM would reject this bytecode because, for safety reasons, all paths leading to a given point have to be equivalent with respect to the contents of the operand stack.

The `ocamljava` compiler performs some code motion to ensure that both semantics are aligned. This means that the code has to be modified in such a way that every time the control flow enters a `try`/`with` construct, the stack is empty. To meet this requirement, it is possible to resort to `let`/`in` constructs, resulting in the following code in the case of our example:

```
let x =
  let tmp = try ... arg1 ... with _ -> ... in
  funct arg0 tmp ... argn
```

where `tmp` is a fresh identifier.

The behavior of this version is equivalent to the previous one, except if the evaluation of `arg0` involves side effects. This is not a fundamental problem, as the OCaml language does not specify the evaluation order. However, developers are used to either left-to-right or right-to-left evaluation order and may find odd to observe any other order. For this reason, the `ocamljava` compiler examines the other arguments to determine whether they may contain side effects. If they do, they are moved just like the expressions containing the `try`/`with` constructs. In our example, if the evaluation of `arg0` cannot be determined to be pure, the following code is actually compiled:

```
let x =
  let tmp0 = arg0 in
  let tmp1 = try ... arg1 ... with _ -> ... in
  funct tmp0 tmp1 ... argn
```

where `tmp0` and `tmp1` are fresh identifiers. The `tmp`*i* variables are stored into the *locals* of the Java method, ensuring that the operand stack remains empty. The code above shows that `ocamljava` evaluates arguments from left to right, because Java method calls expect parameters in this order on the operand stack.

The purity check is currently mainly syntactic, with hard-coded knowledge about some primitives. The check is performed to avoid unnecessary use of local variables that make the code more complex.

The code transformation presented above ensures that there is no semantic mismatch between OCaml and Java exceptions. There is nevertheless another point where exceptions in the two languages differ: speed. OCaml exceptions are notoriously faster than Java ones, so fast indeed that they are sometimes used to manipulate the control flow of a program, just like `break` inside Java loops:

```
try
  for i = ... to ... do
    ...
    if some_condition then raise Exit
    ...
  done
with Exit ->
  ...
```

To get decent performance when exceptions are used, the OCaml-Java runtime generates stack trace elements only when support for backtrace is required.[2] While this greatly improves speed (up to 30% for exception-intensive programs), Java exceptions are still several time slower than OCaml ones.

As a consequence, we introduced an optimized compilation strategy for *static* exceptions. An exception is said to be *static* if we can determine at compile-time which handler will take care of it, should it be raised. For example, in the following code:

```
let f arg0 ... argn =
  try
    ...
    if ... then raise Exit;
    ...
    if ... then raise (Sys_error ...);
    ...
  with
  | Not_found -> ...
  | Exit -> ...
  | _ -> ...
```

it is statically known that if `raise Exit` is executed, the control flow will be passed to the second handler of the `with` clause. Similarly, if `Sys_error` is raised, the control flow will be passed to the third handler. Determining whether an exception is *static* is based on a syntactic criterion. For example, in the following code, the exception raised inside the body of g are not optimized as *static* exceptions because the `raise` expression do not occur in the `try/catch` construct:

```
let g arg0 ... argm =
  ...
  if ... then raise Exit;
  ...
  if ... then raise (Sys_error ...);
  ...

let f arg0 ... argn =
  try
    ...
    g argi ... argj
```

[2] Stack trace generation is disabled by simply overriding the `Throwable.fillInStackTrace()` method with an empty implementation.

```
    ...
  with
  | Not_found -> ...
  | Exit -> ...
  | _ -> ...
```

In practice, this limitation is mitigated by inlining that may decide to replace the g occurrence in f's body by its definition.

The compiler only has to ensure that exception names actually refer to the very same exception and are not homonyms. For example, in the following program, the call to f from g's body will be inlined but the exception E of f is not to be confused with the exception E of g:

```
exception E (* original definition *)
let f x =
  ...
  raise E

exception E (* shadows previous definition *)
let g x =
  try
    f x
  with E -> (* refers to the second definition *)
    ...
```

The compiler also inspects the exception handlers to determine whether the exception value is used; indeed, there is no point in allocating the value if it is never used (it is not uncommon to have *catch-all* clauses such as `try ... with _ -> ...`).

Once *static* exceptions and exception value uses have be determined, respectively in the body and in the handlers of a `try/with` construct, it is possible to turn the `raise`s into bare jumps to the appropriate handlers. The `try/with` construct is however kept, as it can be used to catch exceptions raised by nested calls to other functions.

5.3 Handling of primitives

In the original OCaml implementation, primitives are used as the privileged way to encode manipulations of the runtime system (e.g. creation of blocks), and as the way to access foreign routines through the C interface. The same is true in the OCaml-Java implementation, the foreign routines being obviously written in Java.

Primitives written in the foreign language have to abide by a number of conventions in order to respect the model of the compiled OCaml code. The most obvious convention is of course about data representation, but there are additional constraints in the original OCaml implementation, linked to the garbage collector. Indeed, as memory management is explicit in C, the developer has to indicate to the OCaml runtime which values should be tracked by the garbage collector. In

OCaml-Java, all the code is based on the garbage collector of the JVM whatever its source language is. It is thus not necessary to take care of values that should or should not be tracked by the garbage collector.

The major constraint regarding data representation in the original implementation is that parameter and return values to and from primitives have to be boxed. This may thus imply a runtime cost as the compiler is free to choose an unboxed representation for data manipulated inside an OCaml-written function. The risk is to have to convert back and forth between boxed and unboxed representation at each primitive call. Fortunately, this problem is mitigated by the fact that some primitives are *intrinsics*, meaning that the compiler knows how to compile them directly and hence does not resort to an actual call to the primitive.

In order to get rid of this *primitive wall* hindering performance, the `ocamljava` compiler provides two implementations for the primitives (one with all values boxed, and another one with all values unboxed), when at least one value among the ones passed as parameters or returned can be unboxed. We have to keep two implementations, as the non-optimized one is used for generic application. The optimized implementation takes precedence and is used everywhere else, avoiding to pay the price of boxing/unboxing at every primitive call. The most common case is to have a mix of boxed and unboxed parameters, leading to method signatures akin to `meth(Value,int,float,Value)`.

5.4 Use of propagated types

Beside the *primitive wall* mentioned in the previous section, the original OCaml implementation builds another wall: the *function wall*. This means that, like primitives, functions receive and return values in their boxed form, whatever form is used internally by the function. This problem is indeed more important than the one related to primitives as primitive calls often account for a rather small share of all calls. It is noteworthy though that the problem of the *function wall* can be mitigated in the original OCaml implementation by setting a higher limit to inlining. Unfortunately, the same is not quite true in OCaml-Java as the JIT of the JVM (which is responsible for inlining) tends to prefer smaller methods: the JIT has basically an inlining budget, and will use it to inline several small methods rather than a large one.

The objective is thus to be able, as for primitives, to call OCaml functions accepting and returning unboxed values. It is a bit more difficult for functions than for primitives, as we have to know the types of the manipulated values to known whether the unboxed representation can be used.[3] This explains why, as seen in Section 3, and contrary to the `ocamlopt` compiler, the `ocamljava` compiler attaches type information to lambda code. For example, the following function:

```
let f x y =
```

[3] From a theoretical standpoint, the same is true for primitives but in practice the full type of primitives is set once for all by the runtime implementation.

```
  print_string x;
  y + 10
```

will be annotated in `ocamljava` with the information that `x` has type `string`, `y` has type `int`, and the return value also has type `int`.

By keeping the information produced by the type system, we determine the representation for the parameter and return values of each function. The representation is also saved inside the `.cmj` file as other modules have to also know the representation in order to call the function. It is necessary to store the representation of parameters alongside the module signature, in order to be able to unbox opaque types (e.g. if an opaque type is actually an integer). It would not be enough to store the Java signature either, as a Java long can be used for an OCaml int or an OCaml char.

As for primitives, if any value among the passed and returned ones can be unboxed, two functions are emitted rather than one:

- the *genuine* function, using the unboxed representation;
- an associated *surrogate* function, using the boxed representation and merely performing boxing/unboxing operations to call the *genuine* function.

The second function is used for *generic* application, including partial application, higher-order functions, *etc.*

Beside the types supporting an unboxed representation already presented at Section 4, that are integer and float types, another type enjoys a special treatment: the `unit` type. We indeed erase the `unit` parameters and return values, thus reducing the number of stack operations.

There is another optimization that is performed with the help of type propagation: the use of an optimized representation for blocks used for values of the OCaml `int array` type. In this case, the optimization is not about unboxing the whole value itself but about storing the elements of the block in their unboxed representation. The `ocamlopt` compiler already provides some support for `int arrays`: although the representation is the same as for bare blocks, they enjoy special implementation of element accesses. The key point is that as all elements of the block are guaranteed to be of the `int` type, it is not necessary to register the operation with the garbage collector.

In OCaml-Java, we do not have direct low-level control of the garbage collector, but lessen its work by avoiding unnecessary allocations, and by using unboxed integers. This optimization is performed by rewriting the function calls creating arrays and replacing them with call to specialized functions if the propagated type indicates that the array to be built is made of `int` values. It is noteworthy that this information cannot be retrieved at runtime. For example, suppose the following code is executed:

```
type t = A | B of int
let first_array = Array.make 3 A
let second_array = Array.make 3 0.
```

At runtime, the `Array.make` function will receive two values: the length of the array to build and the initial value for each element. Both calls to `Array.make` receive the very same values, as the value `A` is encoded exactly as the integer `0`. This may lead the function to believe it is building an `int` array in the first call. However, it is not the case and the expression `a.(0) <- B 1` is perfectly legitimate and would try to store a block at index 0. The very limited type information retained at runtime is hence not sufficient, and we have to resort to compile-time information to know whether we are building an `int` array.

5.5 Loop optimizations

The last category of optimizations performed by `ocamljava` with respect to `ocamlopt` are loop optimizations. These basic and well-known optimizations provide some speed improvements. They are all controlled by the *inlining* parameter that can be passed to the compiler through the command-line, as:

- the method size is strictly limited by the Java class file format;
- the JIT does prefer to have to compile several small methods than a big one.

The user can get oversized methods if *inlining* parameter is too high. However, the compiler will not produce an invalid class file but output a meaningful error message.

A `while` loop, as well as a `for` loop whose bounds are unknown, can be optimized by performing "unrolling": jumps are avoided, and references to the loop index are replaced by constants. This means that, according to the inlining threshold and the size of the body, its contents is repeated. For example, the following loop:

```
while cond do
  body
done
```

can be transformed into:

```
while cond do
  body
  if not cond then jump to loop_end
  body
done
loop_end:
```

Loop unrolling can yield better performance by avoiding jumps. It also interacts with partial evaluation, that can further optimize the loop if it can be statically determined that condition is true at start, leading to code akin to:

```
loop_start:
  body
  if not cond then jump to loop_end
  body
  if cond then jump to loop_start
loop_end:
```

A `for` loop whose bounds are known can be completely inlined in such a way that the loop actually disappears, provided the inlining budget is sufficient. For example, the following loop:

```
for i = 1 to 4 do
  let x = i * ... + ... in
  func x
done
```

can be transformed into:

```
let x1 = 1 * ... + ... in
func x1;
let x2 = 2 * ... + ... in
func x2;
let x3 = 3 * ... + ... in
func x3;
let x4 = 4 * ... + ... in
func x4;
```

where the `x`*i* are fresh identifiers. Such a transformation does not only saves costly jumps; it also leverages the combined benefits of inlining and constant folding. This explains why loop unrolling is interesting at the compiler level even though the JVM is also able to unroll loops. Indeed, by performing loop unrolling at the compiler level, we are able to trigger other compiler optimizations such as constant folding, and partial evaluation.

6 Bytecode optimization

The OCaml-Java distribution features a tool named `ocamljar` that is best described as a "*post-compilation optimizer*". It takes as its input a standalone `.jar` file compiled and linked by the `ocamljava` compiler and outputs an equivalent optimized `.jar` file, performing a number of global and local optimizations. The range of optimizations to apply can be selected through command-line switches, and the safety of the optimizations depends on how the program will be used and cannot be ensured by the tool itself. The tool works by transforming class files from the archive, changing fields' definitions as well as method bytecode. It also uses the information saved by the `ocamljava` compiler as Java annotations in the various produced class files.

A Java archive produced by the linking of several previously compiled modules contains essentially the following entries:

- a `Module.class` containing the code for the module, along with `Module$Global` to store global variables and `Module$Constants` to store the module constants;
- an `ocamljavaMain.class` acting as the entry point of the whole application, and that is responsible for calling the `entry()` method of each linked module;

- classes under the `org/ocamljava/runtime` path, forming the OCaml-Java runtime support (written in Java, and compiled by the classical `javac` compiler).

The `ocamljar` tool applies different optimizations to the different kinds of entries, but it is noteworthy that it optimizes both class files produced by the `ocamljava` and `javac` compilers. This is indeed one of the reason for using a post-compilation tool rather than options to the `ocamljava` compiler. It is far more convenient to add a step to the build process of an application than to have to handle multiple versions of the compiled modules and runtime.

The following sections present the main optimizations performed by `ocamljar`. Using the tool can lower startup time by several tenths of seconds, and results in a global performance gain comprised between 5% and 10% on most applications. The actual gain heavily depends on the application profile.

6.1 Optimization of critical sections

As previously mentioned, the original OCaml implementation relies on a global runtime lock that is indeed mandatory as some core runtime routines are not re-entrant and the garbage collector uses a stop-the-world collector. By contrast, the OCaml-Java runtime is re-entrant, and the garbage collector of the JVM can be executed concurrently. There is hence no need for OCaml-Java to be based on a global runtime lock. However, some OCaml programs and libraries make the assumption that the code executes under a global runtime lock. For this reason, the default mode of OCaml-Java is to use such a lock. The `ocamljar` tool allows eliding the lock, the developer being responsible for ensuring that the code is still correct if the assumption of a global lock is lifted.

Throughout the OCaml-Java code (both code from the runtime and code compiled using `ocamljava`), all critical sections materializing the global runtime lock are implemented by calls to methods of the `CurrentContext` class. It is thus sufficient to erase these calls to ensure that the program never acquires the runtime lock; for example, a routine from the i/o support library is transformed from:

```
try {
  CurrentContext.enterBlockingSection();
  ((Channel) channel.asCustom()).write8u(...);
  CurrentContext.leaveBlockingSection();
} catch (final IOException ioe) {
  CurrentContext.leaveBlockingSection();
  Sys.sysError(null, ioe.toString());
}
```

into:

```
try {
  ((Channel) channel.asCustom()).write8u(...);
} catch (final IOException ioe) {
```

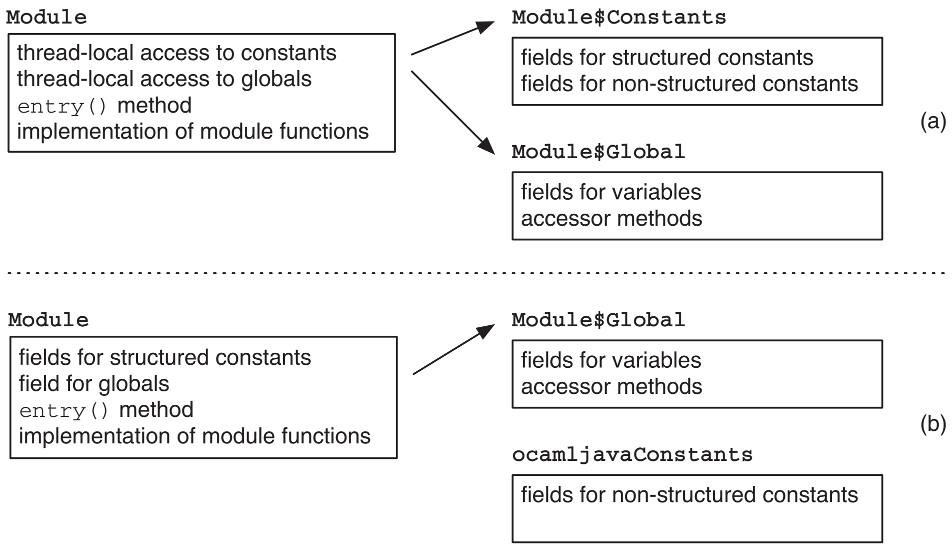
Module
thread-local access to constants
thread-local access to globals
entry() method
implementation of module functions
Module$Constants
fields for structured constants
fields for non-structured constants
Module$Global
fields for variables
accessor methods
(a)
Module
fields for structured constants
field for globals
entry() method
implementation of module functions
Module$Global
fields for variables
accessor methods
(b)
ocamljavaConstants
fields for non-structured constants

Fig. 5. Handling of constants and globals (a) before and (b) after optimization.

```
  Sys.sysError(null, ioe.toString());
}.
```

This optimization is applied to `ocamljava`-produced classes, as well as to the runtime classes.

This optimization is safe if and only if either (i) the program does not spawn threads, or (ii) the program does not assume that OCaml primitives (e.g. printing functions) are protected by a lock.

6.2 Optimization of constants and globals

Constants come in two flavors: non-structured ones that are used for boxed values of type `int32`/`int64`/`nativeint`/`float` and are immutable, and structured ones that are used among other things for values of type `string`/`array` and are mutable. This means that non-structured constants can be safely shared by different modules, while structured ones have to be private to one module (and even to one module *instance* if several programs run on the same JVM).

Figure 5 shows how constants are optimized by the `ocamljar` tool, the arrows representing references to instances. The class used to store constants is dismissed and its fields are moved to the main module class for structured constants and to a newly introduced `ocamljavaConstants` for non-structured ones. The `ocamljavaConstants` also merges constants referring to the same value from different modules. Additionally, the indirection *via* thread-local storage is removed. In practice, this means that an access to a constant is just an access to a static field after optimization.

Global variables still have to be stored in a dedicated instance that should also be a `Value`, in order to be used as a module representation for functor application or first-class module wrapping. However, `ocamljar` performs another kind of optimization that discards the code used to initialize global values that are never read.

Determining which global values are never read of course requires whole-program information. Constructing this information is done in three steps:

1. at compile-time, `ocamljava` stores for each class the list of global elements used (both in the current and in foreign modules);
2. at link-time, `ocamljava` stores the list of modules used to build the application;
3. at optimization-time, `ocamljar` gathers the information produced at the previous steps and builds a complete representation of global variables use.

Equipped with this information, `ocamljar` then removes the code initializing these never-used global variables. A large part of these variables consists of closures corresponding to functions exported by the module. It is crucial not to build these closures as it is a costly operation that requires resolving method handles. Avoiding these closure constructions significantly reduces startup time. It should be noticed that this optimization would not benefit the `ocamlopt` native compiler as it is able to directly store closures in the data segment of the produced binary, thus paying no runtime penalty for their initialization.

This optimization is applied only to `ocamljava`-produced classes.

This optimization is safe if and only if there is one `ocamljava`-compiled program running on the JVM.

6.3 Optimization of context accesses

We want to be able to use `ocamljava`-compiled code to execute applets or servlets. It is also possible to use compiled code as a plugin of Java applications. In such situations, it is possible to have several `ocamljava`-compiled programs running in the same JVM and to share the same modules (and hence class files).

To ensure that the states of the various programs are not intertwined, it is necessary to clearly separate the data of each program. To this end, the *context* of a given program (e.g. current path, the list of opened file channels) is stored as a Java instance and is retrieved from a thread-local variable (a given thread being *attached* to a single program).

However, even if we have to take into account situations where several programs coexist in the very same JVM, the user may also want to get rid of the thread-local indirection when she is absolutely sure that only one program will be run by the JVM. In this case, the `ocamljar` tool is able to rewrite accesses to the context from thread-local value retrieval to simple lookup of a static field.

The first versions of OCaml-Java explicitly passed the context as an additional argument to every function, but it turned out to be quite costly (we gained about 10% when switching to thread-local storage) and most of the time unnecessary because most functions do not use the context.

This optimization is applied to `ocamljava`-produced classes, as well as to the runtime classes.

This optimization is safe if and only if there is one `ocamljava`-compiled program running on the JVM.

6.4 Optimization of array accesses

We have seen in Section 4 that we can partially avoid array bound checks by providing classes specialized for a given size. However, the cost of array bound checks has still to be paid for classes whose size is above that threshold. It is nevertheless common to run production code at full speed, trading safety for speed by removing array bound checks from a well-tested program.

To this end, `ocamljar` can be instructed to change every use of the `BasicBlockValue`, `DoubleArrayValue` and `LongBlockValue` classes to their *unsafe* equivalents. These *unsafe* classes are based on the `sun.misc.Unsafe` class that provides accesses to array elements through pointer arithmetic, avoiding array bound checks.

This optimization is disabled by default as bogus code can corrupt the memory and crash the JVM. Moreover, executing an application based on the `sun.misc.Unsafe` class involves placing its code in the boot class path[4], as privileged rights are mandatory for Java code using unsafe features. Additionally, the use of the unsafe containers does not always result in a performance gain as it may disable some JIT optimizations.

This optimization is applied to `ocamljava`-produced classes, as well as to the runtime classes.

This optimization is safe if and only if there is no out-of-bounds array access in the program.

7 Benchmarks

In this section, we assess how the version 2.0-alpha3 of OCaml-Java[5] compares to the original OCaml implementation. We first focus on CPU performance, and then discuss the accessory elements that are startup time and memory consumption.

7.1 Single core benchmarks

In order to evaluate the performance level of OCaml-Java, we compare it to the original `ocamlopt` compiler. To perform the comparison, we use programs provided by the Computer Language Benchmarks Game[6]. We retain only the programs that do not rely on external libraries (such as GMP). It is important to notice that we used the very same OCaml code for both `ocamlopt` and `ocamljava` compilers, even if the OCaml code present in the Computer Language Benchmarks Game has allegedly been optimized for the `ocamlopt` compiler.

[4] Through one of the following `java` switches: `-Xbootclasspath/a:`*`class-path-elements`*, or `-Xbootclasspath/p:`*`class-path-elements`*.

[5] Available at `http://www.ocamljava.org`.

[6] `http://benchmarksgame.alioth.debian.org`

The various programs exercise different areas of the code generation:

- *binarytrees*, exercising recursive functions over trees;
- *fannkuch*, exercising integer computations over arrays;
- *fasta*, exercising float computations, string manipulations and i/o operations;
- *mandelbrot*, exercising float computations and i/o operations;
- *meteor*, exercising integer computations over arrays;
- *nbody*, exercising float computations;
- *revcomp*, exercising string computations and i/o operations;
- *spectralnorm*, exercising heavy float computations over arrays;
- *threadring*, exercising threads and mutexes.

Additionally, we add the following test cases:

- *meteors*, that is just the repetition (64 times) of *meteor* because *meteor* is so short that start-up has a major impact on overall performance;
- *kb* benchmark from the test suite of the original OCaml distribution, as an example of functional code extensively using higher order functions, and exceptions for backtracking.

The various pieces of software used throughout the benchmark procedure are:

- `java` version`1.7.0_71` build;
- `ocamlopt` version `4.01.0`;
- `ocamljava` version `2.0-alpha3`.

Following the setup of the Computer Language Benchmarks Game, we always pass both `-noassert` and `-unsafe` flags to the compilers. For each compiler, we test the binaries produced with either `-inline 100` (inlining used by the Computer Language Benchmarks Game), or `-inline 0` (inlining being disabled). Moreover, we also assess the impact of the `ocamljar` utility by applying it to each `ocamljava`-compiled program. We hence test six programs:

- `opt+i`: `ocamlopt ... -inline 100`;
- `opt-i`: `ocamlopt ... -inline 0`;
- `ocj+i-o`: `ocamljava ... -inline 100`;
- `ocj+i+o`: `ocamljava ... -inline 100` with `ocamljar` applied;
- `ocj-i-o`: `ocamljava ... -inline 0`;
- `ocj-i+o`: `ocamljava ... -inline 0` with `ocamljar` applied.

Each program is run under Mac OS X seven times, the worst time being ruled out to take into account code warm-up. All programs running on a JVM use the very same parameters, that are `-server -XX:+TieredCompilation -XX:+AggressiveOpts`. Table 2 presents the results for each compiler/benchmark combination with the mean time given in seconds. Figure 6 presents the ratio of the best `ocamljava`-compiled program over the best `ocamlopt`-compiled program for each bench mark.

Table 2. *CPU performance of the various compilers (time in seconds)*

Test	`opt+i`	`opt-i`	`ocj+i-o`	`ocj+i+o`	`ocj-i-o`	`ocj-i+o`	best `opt`	best `ocj`
kb	0.89	0.93	14.66	15.34	13.53	15.40	0.89	13.53
binary trees	29.91	29.83	23.80	19.47	23.20	23.35	29.83	19.47
fannkuch	67.05	76.10	108.37	108.55	107.38	107.46	67.05	107.38
fasta	9.23	9.82	139.66	150.31	15.45	11.18	9.23	11.18
mandelbrot	44.20	43.85	42.34	40.88	35.33	41.19	43.85	35.33
meteor	0.74	0.77	2.89	2.39	2.90	2.41	0.74	2.39
meteors	46.45	48.61	117.72	90.46	112.86	89.61	46.45	89.61
nbody	11.67	11.66	13.01	13.04	12.60	12.57	11.66	12.57
revcomp	3.27	3.28	25.45	17.47	25.39	17.49	3.27	17.47
spectralnorm	11.81	19.54	15.81	15.02	16.33	16.34	11.81	15.02
threadring	61.95	61.89	53.00	50.53	52.94	50.31	61.89	50.31

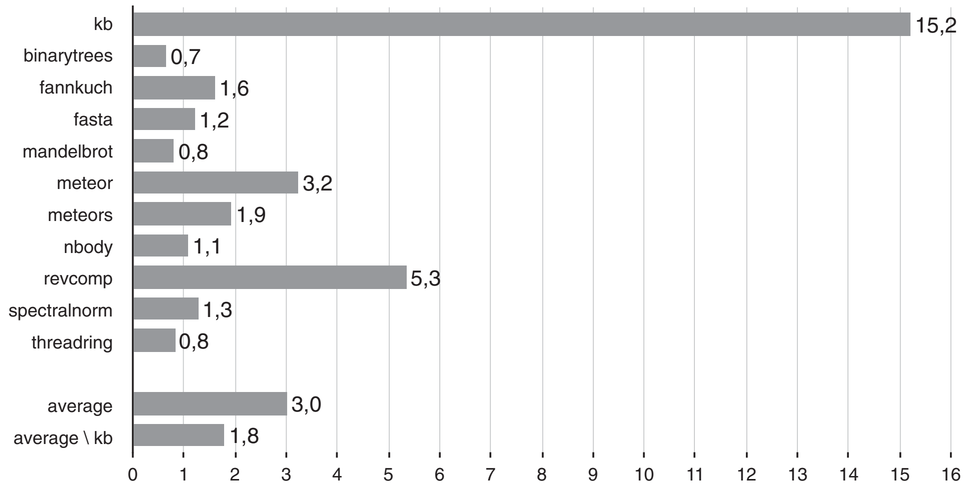

Fig. 6. Ratio of `ocamljava` over `ocamlopt` (using best times).

The performance level of the `ocamljava` compiler is encouraging; on most benchmarks of the Computer Language Benchmarks Game, performance is within a factor of 2:

- *binary trees*: 0.7;
- *fannkuch*: 1.6;
- *fasta*: 1.2;
- *mandelbrot*: 0.8;
- *meteors*: 1.9;
- *nbody*: 1.1;
- *spectralnorm*: 1.3;
- *threadring*: 0.8.

However, the ratio degrades to 3.2 for *meteor* and to 5.3 for *revcomp*. Regarding *meteor*, as shown by *meteors*, the problem is largely due to the shortness of the benchmark that mechanically increases the impact of the start-up time. Regarding

Table 3. *Impact of `ocamljar` (gain in percents)*

Test	`ocamljar` gain (inlining on)	`ocamljar` gain (inlining off)
kb	-4.7%	-13.8%
binarytrees	18.2%	-0.6%
fannkuch	-0.2%	-0.1%
fasta	-7.6%	22.7%
mandelbrot	3.4%	-16.6%
meteor	17.1%	16.6%
meteors	23.2%	20.6%
nbody	-0.2%	0.2%
revcomp	31.4%	31.1%
spectralnorm	5.0%	-0.1%
threadring	4.7%	5.0%

revcomp, a detailed analysis reveals that the bad performance stems from the combination of two elements: string operations are slower in the `ocamljava` runtime, and the unboxing heuristics are not as efficient as in other benchmarks.

Finally, the worst result is obtained on the *kb* benchmark with a ratio of 15.2. This underlines that there is still a lot of work to be done in the area of higher order functions and closures, and that despite some optimizations, exceptions are still far more costly on the JVM. Regarding higher order function and closures, it is important to notice that their use implies that function calls are not *direct* anymore, but are executed through a *generic* mechanism. This mechanism does not only add an indirection, but also requests all parameters to be passed using a boxed representation. Regarding exceptions, it is noteworthy that exceptions were designed in Java to handle errors, while they are sometimes used in OCaml to encode the control flow of the program.

7.2 Impact of post-compilation optimization

Post-compilation optimization, as performed by the `ocamljar` tool has quite an impact on the performance level. Table 3 presents the gain due to `ocamljar` for the various benchmarks. The programs that benefit the most from `ocamljar` are those making frequent use of global variables, including those where such uses are made from functions of the standard library (e.g. a function that outputs strings on the standard output uses the global variable holding the channel associated with the standard output).

7.3 Startup time and memory consumption

CPU usage is not the only metric that can be used to characterize an execution. Memory consumption, and to a lesser extent startup time are other important characteristics. The startup time is measured by executing an *empty* program under the same conditions as before. Results for the various compilers, as well as for Java and Scala, are given in Table 4. The `ocamlopt` compiler clearly outperforms other

Table 4. *Startup time for the various compilers (in seconds)*

Compiler	Startup time
`ocamlopt`	0.01
`ocamljava`	0.32
`ocamljava` (with `ocamljar` applied)	0.30
`javac`	0.13
`scalac`	0.13

Table 5. *Memory consumption on the various benchmarks (in hundreds of megabytes)*

Test	`javac`	`ocamlopt`	`ocamljava`	`scalac`	`ocamljava`/ `javac`	`ocamljava`/ `scalac`
binary trees	16.74	9.71	30.27	16.69	1.8	1.8
fannkuch	0.90	0.03	14.48	1.03	17.2	15.1
fasta	0.93	0.12	2.40	1.17	2.6	2.0
mandelbrot	0.91	0.12	2.26	1.06	2.5	2.1
meteor	1.40	0.20	7.58	4.76	5.4	1.6
meteors	2.12	0.20	13.45	4.77	6.4	2.8
nbody	0.90	0.03	1.56	1.11	1.7	1.4
revcomp	24.98	9.01	33.50	22.01	1.3	1.5
spectralnorm	0.93	0.07	1.57	1.06	1.7	1.5
threadring	6.51	0.30	6.55	...	1.0	...

compilers, which is expected as produced code does not need a virtual machine. Programs compiled with `javac` and `scalac` exhibit similar startup times, while `ocamljava` is 2.3 times slower. The `ocamljava` overhead comes from the initialization of the runtime system, including initialization of cached values and loading of numerous class definitions.

For each benchmark, Table 5 gives the maximum amount of memory used by the process in hundreds of megabytes. We do not differentiate runs with inlining and post-compilation optimization on/off as they only have a minor impact on memory consumption. Unsurprisingly, `ocamlopt`-compiled code uses far less memory than programs run on a Java Virtual Machine. Also, `ocamljava`-compiled code requires in general twice as much memory as `javac`-/`scalac`-compiled code, with two particular benchmarks (namely fannkuch and meteor) where the ratio is far bigger. This can be explained by the fact that a number of OCaml values are boxed, e.g. when stored in a data structure.

8 Related work

In this section, we review projects aiming at providing support for functional languages on *alternative* runtimes. We first describe such projects in the OCaml ecosystem and then consider functional language implementations targeting the JVM, and finally other *managed* runtimes.

8.1 In the OCaml world

We have already mentioned some up-to-date alternative runtimes for OCaml: `js_of_ocaml` (Vouillon & Balat, 2014) that generates JavaScript code, and `ocamlcc` (Mauny & Vaugon, 2012) that generates C code. Both projects are based on the output of the `ocamlc` compiler, the OCaml bytecode being considered as more stable than the internal compiler representation manipulated by `ocamlc`/`ocamlopt`/`ocamljava`. However, this implies that those projects have to reconstruct, either completely or partially through heuristics, information that is explicit inside the compiler and only implicit in the bytecode.

Both projects exhibit good results in terms of performance, and the `js_of_ocaml` project is also well integrated with the JavaScript runtime, supporting for example manipulation of DOM elements to dynamically modify a web page. The OCaml-Java project took a different path by forking from the original compilers at a point where a higher level representation of the code is available. This alleviates the need to recover information, and also allows us to propagate some type information, which has become crucial in recent versions to reach decent performance. It is also enables an extension of the type system that allows the developer to manipulate Java instances from pure OCaml sources (Clerc, 2013b).

There is also an older project worth mentioning, namely OCamIL (Montelatici *et al.*, 2005), that targeted the .NET runtime, which is very similar to the JVM. This project was a patch against the original OCaml distribution, providing a compiler producing MSIL bytecode. The project also included a tool named O'Jacaré.net (Chailloux *et al.*, 2007) responsible for generating wrappers for .NET libraries for the OCaml languages by both direct calls and registered callbacks. One advantage of OCaml-Java over OCamIL is that its runtime library covers almost the whole original OCaml library, while OCamIL lacks supports in various areas.

Another version of the O'Jacaré (Chailloux & Henry, 2004) tool is also available to interface with Java rather than with .NET. In this case, the camljava project (Leroy, 2004) is used to bridge the gap between OCaml and Java through the use of JNI and the OCaml FFI based on C. The O'Jacaré/camljava duo is thus very different from the OCaml-Java project in that it relies on `ocamlc`/`ocamlopt` and lacks binary portability as the C code interfacing JNI and OCaml FFI has to be recompiled on each platform. A fork of the O'Jacaré targeting OCaml-Java has been recently developed, thus lifting the portability issues.

In OCaml-Java, integration with the Java language is made through two different mechanisms:

- an extension to the type system, as already mentioned, to access Java elements from OCaml sources;
- a tool named `ocamlwrap` that takes as input OCaml compiled interface files and generates Java class definitions.

The class definitions generated by the `ocamlwrap` tool allow the Java developer not only to call functions, but also to build values of the various types through a type-safe interface.

8.2 In the Java world

One of the earliest attempts to port a functional language onto the JVM is the SMLj project (Benton *et al.*, 1998; Benton & Kennedy, 1999), that delivers a compiler for the Standard ML language producing Java bytecode. It is noteworthy that this was at the time a greater endeavor than today, as the JVM greatly evolved and nowadays provides all the elements needed to efficiently run foreign languages on the JVM. For example, method handles were not available, and the compiler adds to generate multiple classes to provide support for closures.

The SMLj compiler does not only produce class files to be run on the JVM, it also provides an object-oriented extension to the Standard ML language, supporting creation of Java instances, calls to Java methods, and accesses to Java fields. The language extensions also make it possible to implement Java interfaces with Standard ML code.

This extension of the Standard ML language is quite different from the one we devised for the OCaml language for various reasons. First, we wanted to keep the original syntax in order to be able to reuse every source-level tool already developed for the language. More importantly, the constraints are different as OCaml already possesses an object system, contrary to Standard ML.

More recently, new functional languages appeared on the JVM. Among them, two high-profile languages are Clojure (Hickey, 2008) and Scala (Odersky *et al.*, 2003). These new languages have an advantage over existing ones in that they are by definition not tied by backward compatibility constraints and can thus decide whether to integrate a feature on considerations based on the target platform.

8.3 Elsewhere

Besides the Scala language, OCaml was also an inspiration for the design of the F# language (Syme *et al.*, 2005). The object system is based on the one of the C# language, to allow interoperability between the two languages through the CLR of the .NET platform.

F# programs deliver first-class performance, which is of course made easier by the fact that the language implementers have designed the language given its runtime environment. Moreover, the F# implementers enjoy tail-call support from the .NET platform, while developers implementing functional languages on the JVM do not have access to such a facility.

9 Future work

The future work to be undertaken in the context of the OCaml-Java project can be split into two categories: work related to the compiler itself, and work related to the project at large. The work related to the compiler is obviously focused on performance issues; we list in the following paragraphs some possible enhancements.

First, although unboxing is already quite powerful, it can be enhanced by trying to unbox values besides those of types `int`, `int32`, `int64`, `nativeint`, and `float`. For example, a tuple can sometimes be unboxed by storing each component independently. Inlining can also be enhanced, by inlining recursive functions. This may seems odd at first sight, but if a recursive function has only tail recursive calls and is thus translated into a simple loop, it is then possible to inline it. Another way to strengthen inlining would be to inline higher order functions more aggressively.

Related to higher order functions, function application and particularly partial application could be optimized. The difficulty here is that some OCaml core libraries heavily rely on the actual memory representation of closures, hindering possible Java optimizations related to method handles. It would be possible to rewrite these libraries to pave the way to optimizations in closure representation. Also, as the `ocamljava` compiler already propagates type information up to code generation, this information may be used to monomorphize polymorphic functions.

Another optimization worth mentioning is the intent to let the user choose between 63- and 64-bit representation for the OCaml `int` type. The current implementation favors compatibility and thus uses 63-bit `int` values. However, one alternative OCaml implementation, namely `js_of_ocaml`, uses JavaScript numbers that are only 53-bit `int` values (mantissa of a 64-bit floating-point value), meaning that the developer can overcome such a mismatch. The remaining question is whether third-party libraries rely on the precise representation.

Besides compiler optimizations, some work could also be done in the area of usability. For example, integration of the OCaml and Java language could be tightened by providing a custom interface to libraries such as Swing or JavaFX. Interfacing the latter may imply some modifications to the compiler through the introduction of an additional linking mode in order to handle the JavaFX application model.

Another way to tighten the integration of both languages will be to leverage the novelties of Java 1.8. The most prominent change is the support of so-called *lambdas*. Even if this is mostly a language change (as opposed to a JVM change), it has a decisive impact on the semantic gap between the languages and will thus probably make library sharing easier.

Finally, another area where Java libraries are likely to play a key role is concurrent and distributed programming. The current version of OCaml-Java ships with a comprehensive library for concurrent programming, but it could be extended in order to provide support for distributed programming.

Acknowledgments

Part of this work was performed while the author was visiting OCaml Labs at Cambridge University. The author would like to thank the OCaml Labs for providing a great working environment.

References

Balat, V., Vouillon, J. & Yakobowski, B. (2009) Experience report: Ocsigen, a web programming framework. *Sigplan not.* **44**(9), 311–316.

Benton, N. & Kennedy, A. (1999) Interlanguage working without tears: Blending SML with Java. In Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming, ICFP '99. New York, NY, USA: ACM, pp. 126–137.

Benton, N., Kennedy, A. & Russell, G. (1998) Compiling Standard ML to Java bytecodes. In Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, ICFP '98. New York, NY, USA: ACM, pp. 129–140.

Benton, N., Kennedy, A. & Russo, C. V. (2004) Adventures in interoperability: The SML.net experience. In Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. New York, NY, USA: ACM, pp. 215–226.

Chailloux, E., Canou, B. & Wang, P. (2009) *OCaml for Multicore Architectures*. Available at: `http://www.algo-prog.info/ocmc/web/`.

Chailloux, E. & Henry, G. (2004) O'Jacaré, une interface objet entre Objective Caml et Java. *L'objet*.

Chailloux, E., Henry, G. & Montelatici, R. (2004) Mixing the Objective Caml and C# programming models in the .NET framework. In *Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Oslo, Norway.

Clerc, X. (2007) *The Barista library*. Available at: `http://barista.x9c.fr`.

Clerc, X. (2012a) OCaml-Java: From OCaml sources to Java bytecodes. In *Implementation and Application of Functional Languages*. IFL 2012, pp. 71–85.

Clerc, X. (2012b) OCaml-Java: OCaml on the JVM. In *Trends in Functional Programming*. TFP 2012, pp. 167–181.

Clerc, X. (2013a) OCaml-Java: An ML implementation for the Java ecosystem. In International Conference on Principles and Practices of Programming on the Java Platform, PPPJ 2013, New York, NY, USA: ACM, pp. 45–56.

Clerc, X. (2013b) OCaml-Java: Typing Java accesses from OCaml programs. In *Implementation and Application of Functional Languages*. IFL 2013, pp. 167–181.

Danelutto, M. & Di Cosmo, R. (2011) *Parmap: minimalistic library for multicore programming*. Available at: `https://gitorious.org/parmap`.

Filliâtre, J.-C. & Kalyanasundaram, K. (2011) Functory: A Distributed Computing Library for Objective Caml. In *Trends in Functional Programming*, TFP 2011, pp. 65–81.

Fournet, C., Le Fessant, F., Maranget, L. & Schmitt, A. (2003) JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, Jeuring, J. & Jones, S. (eds), vol. 2638. Berlin/Heidelberg: Springer, pp. 1948–1948. 10.1007/978-3-540-44833-4.

Hickey, R. (2008) The Clojure programming language. In Proceedings of the 2008 Symposium on Dynamic Languages, DLS '08. New York, NY, USA: ACM. Available at: http://dl.acm.org/citation.cfm?id=1408681

Leroy, X. (1990) *The ZINC Experiment: An Economical Implementation of the ML Language*. Technical Report, INRIA.

Leroy, X. (2004) *The Camljava Project*. Available at: `http://forge.ocamlcore.org/projects/camljava`.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. & Vouillon, J. (2013) *The OCaml System Release 4.01. Documentation and User's Manual*. Available at: http://caml.inria.fr/pub/docs/manual-ocaml/

Mauny, M. & Vaugon, B. (2012) *OCamlCC–Raising low-level bytecode to high-level C*. OCaml Users Developers. Copenhagen, Denmark (http://oud.ocaml.org/2012/).

Microsoft. (2000) *The C# language* a collective Microsoft project.

Montelatici, R., Chailloux, E., Pagano, B. *et al.* (2005) Objective Caml on .NET: The ocamil compiler and toplevel. In Proceedings of the 3rd International Conference on .NET Technologies. Open-source project: main developers are Charles Oliver Nutter, Thomas Enebo, Ola Bini and Nick Sieger.

Nutter, C. *et al.* (2008) *JRuby a Java-Powered Ruby Implementation.* Available at: `http://jruby. org`.

Odersky, M. *et al.* (2003) *The Scala Language*. Available at: `http://www.scala-lang.org/`.

Stolpmann, G. (2012) *Plama: Map/Reduce and Distributed Filesystem.* Available at: `http://plasma.camlcity.org/`.

Syme, D. *et al.* (2005) *The F# Language*. Available at: `http://fsharp.org`.

Vouillon, J. & Balat, V. (2014) From bytecode to JavaScript: The js_of_ocaml compiler. In *Software: Practice and Experience*, **44**(8), 951–972.