OCaml-Java: From OCaml Sources to Java Bytecodes

Xavier Clerc(✉)

France
ocamljava@x9c.fr
http://www.ocamljava.org/

Abstract. This article presents the code generation scheme of the OCaml-Java compiler. The goal of the OCaml-Java project is to allow execution of OCaml programs on a Java Virtual Machine. In order to achieve decent performance, it is necessary to build a compiler producing optimized bytecode that will rely on an efficient support library at runtime.

The OCaml-Java project thus provides (*i*) an efficient runtime written in pure Java, and (*ii*) an optimizing compiler based on the original OCaml compilers for the front-end and on the Barista library for the back-end.

Keywords: OCaml · Java · Bytecode · Compiler · Code generation

1 Introduction

The OCaml-Java project has been presented at large in previous work [1]; in the present article, we will focus on the code generation process as implemented in the OCaml-Java compiler. In the remainder of this section, we will nevertheless summarize the goals and state of the OCaml-Java project. Then, Sect. 2 will expose the architecture of the various OCaml compilers. Section 3 will present the runtime representation of values in the different compilers, and Sect. 4 will give an overview of the Barista library that is used as the compiler back-end. Section 5 shows examples of actual bytecode generation, and Sect. 6 shows how the compiler performs on some benchmarks. Finally, Sect. 7 will discuss future work.

Why the JVM is an Interesting Target

The official OCaml distribution features both bytecode (for a dedicated virtual machine), and native compilers (for common architectures and OSes). It may seem at first sight that nothing more is needed, the former meeting portability needs and the latter meeting performance needs. However, being able to run OCaml code on a Java Virtual Machine is appealing for mainly two reasons:

- access to a larger choice of libraries;
- access to multicore programming.

R. Hinze (Ed.): IFL 2012, LNCS 8241, pp. 71–85, 2013.
DOI: 10.1007/978-3-642-41582-1_5, © Springer-Verlag Berlin Heidelberg 2013

The number of available libraries is still a known weakness of the OCaml ecosystem in spite of a vibrant community. Having the ability to run on a Java Virtual Machine gives access to all the libraries of the Java ecosystem. The Java community is huge, and has developed frameworks and tools for almost any purpose. There are obvious benefits for OCaml developers to use these libraries.

To be able to use the Java libraries, it is not sufficient to produce Java bytecode. It is also necessary to give to the OCaml developer means to manipulate Java objects from an OCaml program. For this reason, the OCaml-Java compiler features an extension of the type system to allow the construction and manipulation of Java instances from a pure OCaml program. More details regarding the extensions to the type system can be found in our introductory article [1].

Multicore programming can be done in OCaml without resorting to compilation to Java bytecodes. However, the original implementation of OCaml is based on a global runtime lock allowing only one OCaml thread to run at a time. For this reason, leveraging multiple cores is often done through libraries using indeed multiple processes (most notably, map/reduce implementations [2,3]).

Another option is to modify the OCaml runtime to get rid of the global runtime lock. Such a modification implies of course to develop a parallel garbage collector [4] and needs a lot of manpower, as well as some modifications to core OCaml libraries that are not reentrant. At the opposite, by targeting a Java Virtual Machine, we get a parallel garbage collector for free, and in addition can take advantage of Java standard libraries such as the fork/join framework to develop multicore OCaml programs based upon shared-memory.

Java 1.7 Features for Functional Programming

The latest major release of the Java platform has brought a lot of exciting new features. Among them, two are particularly interesting when implementing functional languages:

- the `invokedynamic` framework;
- the *G1* garbage collector.[1]

The `invokedynamic` framework is a very powerful addition to the Java platform as it allows a language implementor to define new semantics for method dispatch. In the OCaml-Java project, we in fact only use the method handles (which are akin to function pointers in C) provided by the framework in order to easily and efficiently implement closures.

The *G1* garbage collector is actually pretty important for functional language implementors because it is known to better suit the allocation/collection pattern found in functional programs. Such programs are typically allocating a lot of small and short-lived values while classical Java programs tend to put less pressure on the allocator.

[1] Already present in previous version, but not production-ready.

Past and Present of OCaml-Java

The 1.x versions of the OCaml-Java project should be regarded as mere proofs of concept, whose goal was to reach compatibility with the original implementation. The compatibility is almost total: all language constructs are supported and most OCaml libraries exhibit the same behavior (some minor differences are due to the fact that the Java Virtual Machine does not implement all POSIX primitives).

The 2.0 version described in this paper keeps the same compatibility level, and features great improvements in both memory usage and performance. The goal is to be able to execute typical OCaml code on a Java Virtual Machine while remaining at worst two times slower than native code. The current prototype fulfills this objective on the majority of tested benchmarks.

2 Compiler Architecture

Original Compilers

The original OCaml distribution ships with two compilers: one producing bytecode for a dedicated virtual machine, and the other one producing native code. The bytecode compiler is available on every architecture while the native one is only available on the following:

- tier 1 (i.e. officially maintained): `amd64`, `ia32`, `powerpc`, and `arm` under Linux, MacOS X or Windows;
- tier 2 (i.e. unofficially maintained): `sparc`, and tier 1 architectures under BSD or Solaris flavors.

Both compilers naturally share a large codebase: parsing and typing are identical, thus relying on the very same code. Figure 1 shows the successive passes of both compilers from an implementation source file (i.e. a `.ml` file) to an implementation compiled file (i.e. a `.cmo` file for the bytecode compiler, and a `.cmx` file for the native compiler). We do not detail the compilation of an interface source file because it (*i*) does not produce code, and (*ii*) it 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 on standard output to ease 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 produces so-called *lambda code*, which is the most abstract representation of code to be compiled.

From this point, 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 was designed to provide the pieces allowing to almost execute *lambda code*. Of course, the native compiler

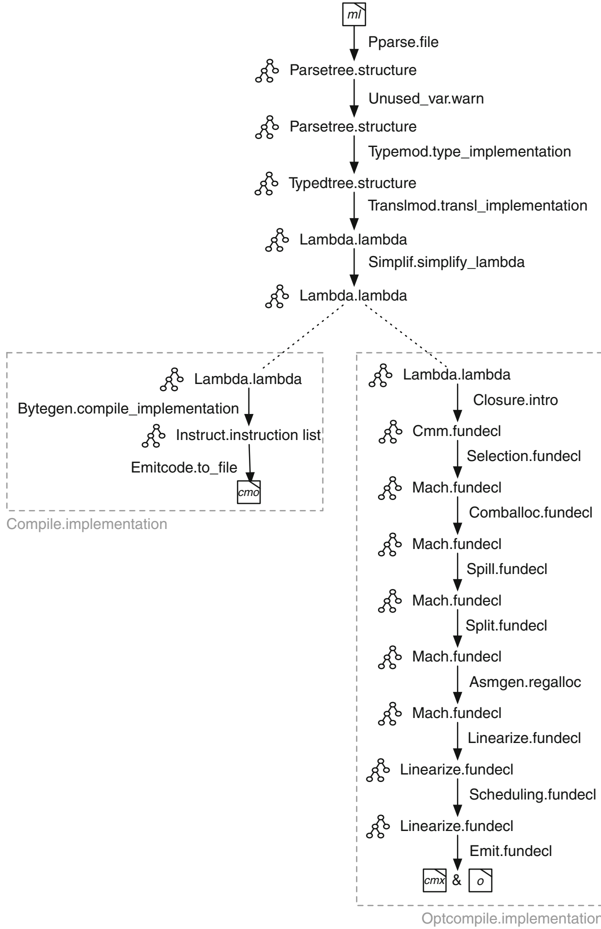

Fig. 1. Passes of OCaml compilers.

has far more work to do 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 step, `Closure.intro`, handles the transformations associated with closures, uncurrification, and related optimizations. From this point, the code is represented by *machine code* which is an abstract representation that is still largely independent from the target platform, based on pseudo-instructions. The `Selection.fundecl` and `Comballoc.fundecl` are designed to perform the selection of pseudo-instructions for the code, and the optimization of allocations linked to a given block. Then, `Spill.fundecl`, `Split.fundecl`, and `Asmgen.regalloc` are responsible for actual register allocation, using information from the target platform. Finally, `Linearize.fundecl` reifies pseudo-instructions into actual lists of instructions, and `Scheduling.fundecl` optimizes the resulting

order. The very last step is to output the assembly source code that will be used by an external assembler to produce object code.

OCaml-Java Compiler

The OCaml-Java compiler can be seen as a third branch of the tree depicted by Fig. 1. This means that passes up to `Simplif.simplify_lambda` are shared with the original compilers. Figure 2 shows which transformations are then made on *lambda code*. First, very similarly to the native compiler, `Jclosure.jlambda_of_lambda` is responsible for the handling of closures, producing a slightly different and optimized *lambda code*. Then, `Macrogen.translate` decomposes operations from the *lambda code* into *macro instructions* that are not Java bytecode instructions but can be easily mapped to. This pass is also responsible for variable allocation which entails the choice of their actual representation, thus opening the possibility of value unboxing. Finally, `Bytecodegen.compile_function` produces actual Java bytecode using the Barista library (detailed at Sect. 4).

The point where native and OCaml-Java compilers diverge (namely `Jclosure.jlambda_of_lambda`) has been chosen because the latter has to be more aggressive regarding constants handling and propagation. Indeed, the native compiler does not need to optimize long values, as they are always unboxed. Another construct is treated in a different way in OCaml-Java: switches because the Java instruction set features both table and lookup instruction while the native code generator only emits code corresponding to table switches.

The next pass of the OCaml-Java compiler (that is `Macrogen.translate`) 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 (based on the initialization value of a variable to determine its type). This compilation pass is also responsible for the handling of exceptions, as there is a mismatch between the OCaml and Java semantics on the subject. The difference is that, in Java, when an exception is thrown the stack is immediately emptied and the instance of the

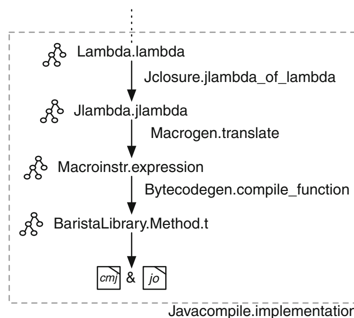

Fig. 2. Architecture of OCaml-Java compiler.

thrown exception is then pushed onto the stack. In OCaml, the raise of an exception will only pop stack values until it finds the enclosing `try/with` construct. As a consequence, we have to do some code motion such that an exception can only be raised at a point where the stack is empty: by enforcing this rule we guarantee that both semantics are actually aligned.

Finally, the last compiler pass, `Bytecodegen.compile_function`, uses the Barista library to build an in-memory representation of the class file to emit. This pass is quite straightforward as boilerplate operations such that the computation of stack maps are handled by the Barista library. Indeed, the only important optimization handled by this pass is the tail call optimization. Whenever a call to a function is to be generated, it is checked whether it is a call to the current function. If so, function parameters are placed into locals, and a jump to the method start is emitted. Otherwise, function parameters are placed onto the stack, and a bare static method call is emitted.

Once compilation is done, two files are produced: a `.cmj` file corresponding to the `.cmx` file of the native compiler, and a `.jo` file corresponding to its `.o` file. The `.jo` file is actually a Java archive containing two entries:

- `Module.class` is the class file containing the implementation of all module functions as Java static methods;
- `Module.consts` is a binary file respecting the OCaml marshal format containing the (structured) constants used by the module.

A module is later linked to produce an executable jar file. At runtime, the initialization code for a module (located in its `entry` method) is responsible for the loading of the constants from the `Module.consts` resource. The constants[2] are then accessed through thread-local storage. This indirection is indeed necessary in order to allow several OCaml programs to run on the very same Java Virtual Machine.

3 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. This is not a problem either for Java interoperability: a Java instance will be wrapped in an OCaml value, but its actual class can still be retrieved at runtime if needed through the mechanism of reflection.

Basically, all values share a common type, 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 trivial.

More precisely, 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

[2] Despite their name, some constants may in fact be modified, hence the impossibility to share them between programs running in the very same Java Virtual Machine.

representation it prefers for local values. This freedom is indeed crucial in order to reach good performance because it allows 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 such a specification is implemented in OCaml-Java.

Original Runtime

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

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

A long value 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. The encoding of an integer value i as a long unboxed value l is thus done according to the following equation: $l = (i \times 2) + 1$. 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 block allocated as parts of the OCaml heap.

managedblock	::=	tag $\oplus$ size $\oplus$ list of *size* blocks
	|	closure-tag $\oplus$ size $\oplus$ code pointer $\oplus$ list of *size - 1* blocks
	|	string-tag $\oplus$ size $\oplus$ array of *size* bytes
	|	double-tag $\oplus$ 64-bit float value
	|	double-array-tag $\oplus$ size $\oplus$ array of *size* 64-bit float value
	|	custom-tag $\oplus$ identifier $\oplus$ size $\oplus$ array of *size* bytes

As seen by the possible contents of a managed block, some typing information seems to be retained at runtime. However, this is not enough to recover the typing information present in the source, because several different types in the source can be mapped to the same runtime representation. Again, strong typing has been enforced at compile time, so no confusion could be made at runtime between values of different types.

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.

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 Java Virtual Machine 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 allows to share values between -128 and 255. These values are allocated once at program startup, and also allow to use reference comparisons for values between the bounds.

The compilation scheme of OCaml will turn 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 will not generate code for testing such bounds. However, in Java it is not possible to remove bounds checks when accessing the elements of an array.[3] 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 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 resorted to what could be called *data inlining*. Rather than having only one class named `BasicBlockValue` storing its elements as one `Value[]` field, we define a bunch of classes named `BasicBlockValue`n that store n elements as n `Value` fields. This allows to defines methods such as `get0()` that will return the first element of a value with no bound check. The same is done for double arrays and allows "small" tuples, records and all types sharing the same runtime representation to avoid bound checks when accessing the element at a given index.

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 to avoid maintenance issues. Of course, besides those classes, a `BasicBlockValue` (respectively a `DoubleArrayBlockValue`) 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 test bounds at runtime for every access.

Alternative Encoding of Values

At first, one may question 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

[3] The Hotspot compiler can remove such tests if it can *prove* that no illegal access will happen, but the developer can not request to remove such tests.

between the various kinds of blocks. Unfortunately, we have to closely follow the encoding of the original compilers because some core libraries of the OCaml distribution 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 very same memory layout in OCaml-Java as in the original compilers.

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.

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 there is an existing reference implementation. Indeed, languages such as Clojure [5] or Scala [6] are completely free to design their encoding scheme because they do not have to abide to an existing specification. At the opposite, projects such as JRuby [7] or OCaml-Java have a more constrained design space. For example, the idea of *data inlining* in order to avoid array bounds checks is also used in JRuby.

4 The Barista Library

Overview

Barista [8], by the same author, is initially an OCaml library designed to load, construct, manipulate, and save Java class files. The library supports the whole class file format as defined by Oracle (formerly Sun) up to version 1.7. On top of the library, a command-line utility (also named "barista") has been developed: both an assembler and a disassembler for the Java platform.

The assembler will turn an assembly source file into a class file to be run on a Java Virtual Machine. The disassembler does the same work in the opposite direction: it takes the fully qualified name of a Java bytecode class file present in the classpath, and transforms it into an assembler source. Two other utilities allow to inspect the contents of a bytecode file: it is possible to just print the list of methods of a given class, and also to print the control flow of a given method as a graph.

While other libraries for bytecode manipulation already existed at the time we started the development of Barista, they were not satisfactory alternatives in our case. The most important thing is that we wanted to generate code through a proper library, and not by invoking an external assembler. The underlying motivation is that we want to use the type system to reject obviously wrong bytecode (e.g. pushing an integer value instead of a float one). When using an external assembler, one generates bare text and even type errors only show up at runtime.

Moreover, Barista is also used in the opposite direction: to load class definitions rather than to produce them. This feature is of utmost importance for the extension of the type system: as we deal with manipulation of Java entities, we need to be able to inspect a class contents at compilation time.

Finally, Barista provides some features that are not available in other bytecode libraries, such as the ability to visualize the bytecode of a given method as an hypergraph, or the ability to create/inspect serialized values.

Hypergraph

Besides the representation of methods as lists of instructions, the code of a method can also be represented as a graph. Precisely, a method code can be represented as a rooted hypergraph. The rooted property stems from the fact that there is only one entry point for a given method. The hypergraph nature of the structure is indeed a design choice that allows to represent the conditionals by edges with one source and as many destinations as there are possible outcomes.

The nodes of the hypergraph are labelled with instruction lists that contain no jump, jumps being represented by edges. Edges hence represent the control flow of the method and can be:

- classical edges with one source and one destination, in order to encode sequential execution (the edge is then with no label);
- three-legged edges with one source and two destinations, in order to encode a test and its two possible consequences (the edge is then labelled with the condition associated with the test);
- n-legged edges with one source and $n - 1$ destinations, in order to encode switch instructions (the edge is then labelled with the definition of the switch, that is either a list of values or lower and upper bounds);
- *special* edges with one source and one destination, in order to indicate that the source is protected by a `try/catch` construct, the destination being the exception handler (the edge is then labelled with the class name of the exceptions that can be caught).

Given the hypergraph structure, there are two kinds of optimizations that can be performed by the Barista library:

- structural optimizations, modifying the hypergraph structure;
- non-structural optimizations, modifying only the labels of nodes.

In the first category, Barista currently features two optimizations: dead code elimination, and jump optimization. Dead code elimination removes all nodes that cannot possibly be reached from the root. Jump optimization short-circuits consecutive jumps with no bytecode between them.

In the second category, Barista features several peephole optimizations that are performed independently on the hypergraph nodes. These include, among others:

– code size optimizations (e.g. replacing a *generic* instruction such as `aload` by a more compact `aload`n);
– removal of unnecessary load and/or store operations (e.g. if a loaded value is discarded or if a stored value is overwritten with no use);
– expression simplifications related to neutral or absorbing elements (e.g. addition to zero);
– basic strength reduction (e.g. shifting rather than multiplying when the multiplier is a power of 2).

Example

As an example, we consider the following Java static method, doing some computation over integer values:

```
public static int meth(final int x, final int y) {
  if (x > y) {
    try {
      return compute1(x);
    } catch (final Exception e) {
      return 0;
    }
  } else {
    return compute2(y);
  }
}
```

After compiling it with the `javac` compiler, we can dump its bytecode by invoking the `javap` utility, leading to the following output:

```
public static int meth(int, int);
  Code:
     0: iload_0
     1: iload_1
     2: if_icmple    13
     5: iload_0
     6: invokestatic  #2             // Method compute1:(I)I
     9: ireturn
    10: astore_2
    11: iconst_0
    12: ireturn
    13: iload_1
    14: invokestatic  #4             // Method compute2:(I)I
    17: ireturn
  Exception table:
     from    to  target type
         5     9    10   Class java/lang/Exception
```

Barista can be used to transform a method bytecode into an hypergraph by executing the `barista flow 'C.meth(int,int):int'` command where `C` is the class defining the method. The result is a graph representation in dot[4] format and is represented in Fig. 3.

Figure 3 features seven graph elements:

– four nodes (represented by rectangular boxes), containing the bytecode for the various code blocks (condition evaluation, if block, else block, and exception handler);

[4] See http://www.graphviz.org/.

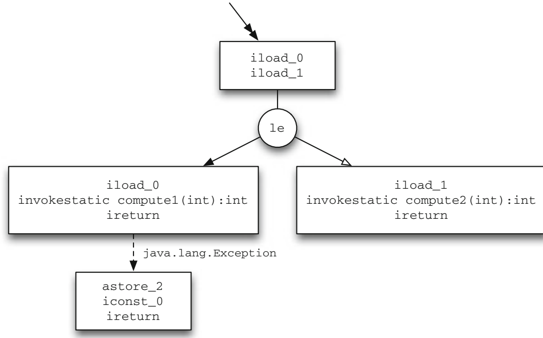

Fig. 3. Hypergraph for method `meth(int,int):int`.

- a double arrow, indicating which node is the root;
- a dotted edge, from the protected node to the handler node and also labelled with the class of exceptions to be caught;
- an hyperedge, linking three nodes: (*i*) the block evaluating the condition, (*ii*) the block to execute next if condition is true, (*iii*) the block to execute next if condition is false; the hyperedge is also labelled with the kind of condition to perform.

5 Example of Bytecode Generation

Our example has been designed to show how the unboxing of values allows to reach good performance in the case of numerical code. The left column shows the OCaml code of the complete function, while the right one shows the generated bytecode for the loop body:

```
let float () =
  let x = ref 1. in
  let y = ref 2. in
  let acc = ref 0. in
  for i = 1 to 1_000_000_000 do
    acc := !acc +. (!x *. !y);
    x := !x +. 1.;
    y := !y *. 2.
  done;
  !acc
```

```
(...)
33: dload 5
35: dload_1
36: dload_3
37: dmul
38: dadd
39: dstore 5
41: dload_1
42: dconst_1
43: dadd
44: dstore_1
45: dload_3
46: ldc2_w 2.0d
49: dmul
50: dstore_3
(...)
```

Variables `x`, `y`, and `acc` are respectively stored at local indexes 1, 3, and 5. The compiler has determined from their initial values that they are double values. Instructions at offsets $33 - 39$ compute the expression `!acc +. (!x *. !y)` and store its value back. Then, instructions at offsets $41 - 44$ update the value of the `x`

variable, and instructions at offsets $45-50$ update the value of the `y` variable. It is obvious from the instructions that all operations are done using the Java `double` primitive type, no boxing is done at all. This ensures that we get the best possible performance, and also avoid to put any pressure on the memory allocator and garbage collector.

When comparing the performance of the original OCaml compiler to the OCaml-Java compiler, we measured the code generated by the former to take 3.8 s and the code generated by the latter to take 5.6 s. Then, we changed the upper bound of the loop by multiplying it by ten, and then measured times to be respectively 38.6 s and 48.0 s. This means that in the second setting, OCaml-Java is less than 25 % slower than original OCaml. Of course, the ratios are better when measuring longer runs because virtual machine startup and just-in-time compiling are amortized.

6 Benchmarks

Procedure

Rather than developing benchmark programs from scratch, we decided to reuse established ones: those from the "Benchmarks Game" (that was previously known as the "Language Shootout"[5]). In order to compare performance between `ocamlopt`- and `ocamljava`-compiled code, we resorted to the following procedure:

– each program is executed 7 times;
– the best and worse times for each program are dropped;
– the remaining times for each program are averaged.

Running the programs several times is of course mandatory to mitigate possible interference from other processes on the testing computer. In the case of performance evaluation for programs running on a JVM, it is also very important to ensure that the virtual machine has been warmed up. This explains why we have to drop the worst execution time (that is, in practice, the first execution time). Finally, it is important to state which options are passed to the JVM: `-server`, `-XX:+TieredCompilation`, and `-XX:+AggressiveOpts`.

Numbers

Table 1 shows the results as ratios (execution time of `ocamljava`-compiled code over execution time of `ocamlopt`-compiled code). The *meteor** program is just the repetition of *meteor* 64 times: the running time for *meteor* is so short that virtual machine startup is significant.

Those results show that the OCaml-Java compiler is on par with the original one on some benchmarks (thread-based and numerical ones), and most of the time between two and three times slower than original OCaml. Given that the

[5] See http://benchmarksgame.alioth.debian.org.

Table 1. Some benchmarks from the Benchmarks Game.

Benchmark	`ocamljava/ocamlopt`	Benchmark	`ocamljava/ocamlopt`
binarytrees	1.75	*nbody*	1.00
fannkuch	3.11	*revcomp*	2.01
mandelbrot	1.58	*spectralnorm*	2.66
meteor	6.81	*threadring*	1.12
*meteor**	4.50		

OCaml-Java compiler is still at prototype stage, and the ability to leverage multiple cores from an `ocamljava`-compiled code, we regard the results as encouraging. Our goal of making OCaml-Java competitive with original OCaml from a performance standpoint seems reachable. However, we clearly need to add new benchmarks to our suite in order to gain more confidence on the preliminary results presented here.

7 Future Work

Most of our short-term effort will be focused on the unboxing of values. It proved to produce large speedups in the past, and a lot of things can be done to make it more aggressive. First, currently, the kind of storage is chosen according to the initial value of a variable; we could design an heuristic also based on the uses of the variable. Second, as previously said, boxing is mandatory at function boundaries; there are two ways to lift this restriction: (i) avoid such a boundary (e.g. by using inlining) or (ii) allow the compilation to functions taking unboxed parameters when typing information allows to do so. Also, unboxing is currently done only for the following OCaml types: `int`, `int32`, `int64`, `nativeint`, and `float`. It could also be done on others types, particularly ones constructed (`e.g.` records with mutable fields) over those that can already be unboxed.

Inlining itself can also be greatly improved. For example, the current version of the compiler is unable to inline recursive functions. This seems like a reasonable limitation at first, but some recursive functions can be tail-call optimized and thus be compiled as mere loops. In this case, it would be possible to inline such functions.

Another area we should definitely investigate is the possible influence of garbage collection parameters over performance. It would have had little sense for the example presented in this paper, but we expect performance to be sensitive to garbage collector parameters in real-world applications. Indeed, the default parameters are chosen to allow good performance for typical Java applications, not OCaml ones. The former ones tend to use big and long-lived instances, while the latter ones tend to use small and short-lived instances.

Finally, we could also optimize compile-time performance by generating the Barista hypergraph directly during code generation. Currently, the compiler produces plain bytecode that is then passed to Barista for low-level optimizations. This incurs the price of hypergraph construction from a list of bytecode instructions, which can be avoided.

To conclude, some words about optimization opportunities that are linked to the future development of the Java platform. Among those considered for inclusion in the next revision of Java, two would be particularly useful to functional languages targeting the Java Virtual Machine. The first feature is tagged values, and would allow us to avoid boxing of `int` values: it would not only allow faster operations but would also relieve the pressure over garbage collection by avoiding allocation. The second feature is support for tail calls, and would allow us to mark a method call as terminal to indicate to the *just-in-time* compiler that a call can be optimized. It would allow, of course, faster execution, but would also make the life of users easier because the absence of tail call optimization interacts with semantics when calls come to blow up the stack.

References

1. Clerc, X.: OCaml-Java: OCaml on the JVM. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 167–181. Springer, Heidelberg (2013)
2. Danelutto, M., Di Cosmo, R.: Parmap: minimalistic library for multicore programming. https://gitorious.org/parmap
3. Stolpmann, G.: Plama: Map/Reduce and distributed filesystem. http://plasma.camlcity.org/
4. Chailloux, E., Canou, B., Wang, P.: OCaml for Multicore Architectures. http://www.algo-prog.info/ocmc/web/
5. Hickey, R.: The clojure programming language. In: Proceedings of the 2008 Symposium on Dynamic Languages. DLS '08, pp. 1:1–1:1. ACM, New York (2008)
6. Odersky, M., et al.: The Scala Language. http://www.scala-lang.org/
7. Nutter, C.O., et al.: JRuby. http://jruby.org
8. Clerc, X.: The Barista library. http://barista.x9c.fr