OCaml-Java: OCaml on the JVM

Xavier Clerc – forum@x9c.fr

August 1, 2012

Please notice *This work will be published and presented at TFP 2012*[1]

Abstract This presentation introduces the OCaml-Java project whose goal is to allow compilation of OCaml sources into Java bytecodes. The ability to run OCaml code on a Java virtual machine provides the developer with means to leverage the strengths of the Java ecosystem lacking in the OCaml world. Most notably, this includes access to a great number of libraries, and foundations for shared-memory concurrent programming.
In order to achieve this, the OCaml-Java project does three contributions: *(i)* an optimized compiler and runtime support to achieve acceptable performances, *(ii)* an extension of the classical OCaml typer to allow manipulation of Java elements from the OCaml world, and *(iii)* a library dedicated to concurrent programming.

Why add a Java backend to OCaml?

The official OCaml distribution already ships with a compiler to OCaml bytecode, as well as a bunch of native compilers for various architectures (mainly `amd64`, `ia32` and `powerpc`, but also `ia64`, and `arm`). It is thus legitimate to question the need for a new backend: the official compilers seem to cover the spectrum from portability (through dedicated bytecode and virtual machine) to performances (through native compiler).

Our understanding is that a Java backend is nevertheless interesting for at least three reasons:

- access to Java libraries, frameworks, and infrastructure;
- access to a multicore facility;
- access to a *secure* environment (in the sense of the Java security manager).

Overview of OCaml-Java

The OCaml-Java project consists of three parts:

- the actual compiler, acting as its central component;
- a runtime support comprising elements needed by compiled code to run (*e. g.* representation of values), as well as a port of the standard library primitives (that were originally written in C);
- a library that is specific to OCaml-Java (*i. e.* that cannot be used with the original implementation), and provides support for concurrent programming.

Typer Extensions

When facing the necessity to provide interoperability between languages, several choices can made such as:

- using an exchange language (*e. g.* XML, JSON, *etc.*);

[1] `http://www-fp.cs.st-andrews.ac.uk/tifp/TFP2012/TFP_2012/Home.html`.

- using bridge generators (*e. g.* based on an IDL);
- embedding the typing of one language into the other one.

The limitations of the first two options led us to think that embedding the (part of the) typing of Java elements into the OCaml compiler would indeed provide the simplest interface to the developer. The obvious goal underlying the extension of the OCaml typer is to make it possible, if not easy, to manipulate Java elements from OCaml code.

We first decided to represent Java instances with a polymorphic abstract type that will enjoy special typing rules. This decision is indeed quite important in its own right, because it clearly indicates that we will not rely on the object system of OCaml to encode the Java class hierarchy. This stems from the fact that the object models of the two languages are really different, and attempts to encode one into another (such as Nickel[2], or O'Jacare[3]) lead to awkward OCaml class definitions and difficult-to-understand error messages.

As a consequence we define a type `'a java_instance` that represent instances of class `'a`. As soon as we do so, we have to introduce a first modification in order to designate, for example, instances of class `java.lang.String`. Indeed, it is not possible to write the type `java.lang.String java_instance` and we decided that Java name classes should be written in OCaml types with simple quotes rather than dots to separate their various parts, leading to `java'lang'String java_instance` which is a perfectly legitimate OCaml type.
Of course, this implies that the elements *under* the `java_instance` constructor are not treated as regular OCaml types, but as special elements designating Java classes.

Further typer extensions allow direct creation and manipulation of Java elements from the OCaml language without any need for new syntactic elements. This includes accesses to methods, constructors, fields, as well as implementation of Java interface through OCaml code. Keeping the very same syntax is indeed a good property, because one can thus still use all the tools that are designed to work on a syntax level.

Performances

The current prototype has been compared to the OCaml native back-end (under `amd64`) on several medium-sized benchmarks from the Language Shootout[4].

Preliminary verdict is quite encouraging. The ratio of `ocamljava` to `ocamlopt` varies from 0.96 to 7.14, and is below 3 in six benchmarks among eight. When looking at the benchmarks producing the worse results, it becomes clear that `ocamljava` is less competitive when computation is done over `int` values. This is not surprising, as such values are always unboxed in `ocamlopt` (using a bit to distinguish them from pointer values) while other values are boxed. Due to the lack of support for tagged values in the JVM, `ocamljava` boxes all values.
Other benchmarks perform intensive computations on values that are boxed in both `ocamlopt` and `ocamljava`, allowing the latter to be on average less than two times slower than the former.

The average ratio is thus very promising, and we want to stress the fact that OCaml code running on a JVM can already exploit shared-memory concurrency, through a dedicated library shipped with OCaml-Java. This allow the developer to leverage the power of multicore computers, favoring *implicit* concurrency (fork/join, parallel array operations, *etc.*).

[2] `http://nickel.x9c.fr/`
[3] `http://www.pps.jussieu.fr/~henry/ojacare/`
[4] `http://shootout.alioth.debian.org/`