OCaml-Java: Typing Java Accesses from OCaml Programs

Xavier Clerc
ocamljava.org
ocamljava@x9c.fr

Abstract

Functional languages, although they often enable great developer productivity and ease software maintenance, are commonly hampered by smaller communities and fewer industrial-strength players when compared to *mainstream* languages. To partially overcome these problems, it is usual to resort to a form of interoperability that allows a functional language to take advantage of libraries originally written for another language.
Initially driven by practical needs, the problem of language interoperability is also fertile in that it induce the designer to develop a better understanding of the differences, and respective strengths and weaknesses of the languages involved.
In this article, we present an extension of the OCaml type system that allows the developer to manipulate Java instances inside OCaml programs. This extension is essentially based on existing features of the OCaml type system, leveraging phantom typing, subtyping, polymorphic variants, and value-directed typing to encode Java types. Combined to this encoding, some lightweight notations allow the developer to easily create and manipulate Java instances from pure OCaml code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages

Keywords Compiler, type system, Java, OCaml, interoperability

1. Introduction

The goal of the OCaml-Java project, broadly presented in [6] is to allow seamless integration of the OCaml and Java languages. The incentives for combining these languages can be split into two categories:

- *Java as a language:* using Java libraries from the OCaml language;
- *Java as a platform:* using facilities available to to Java bytecode.

Leveraging Java as a language is useful to OCaml developers in order to gain access to industrial-strength libraries such as GUI frameworks, or connection layers to database systems. Leveraging Java as a platform is useful to OCaml developers in order to gain access to *alternative* executing modes such as applets, or servlets.

It is noteworthy that some use cases leverage both Java as a language, and as a platform. For example, to develop multicore programs:

- the Java platform is used to benefit from a parallel garbage collector;
- the Java language is used to benefit from fork/join computation, compare-and-set primitives, *etc*.

Indeed, multicore programming is one of the objective of the OCaml-Java project. The original OCaml implementation features a garbage collector that is celebrated for its efficiency, but is based on a global lock that forbids several threads to run OCaml code at the same time. By compiling OCaml sources to Java bytecode in order to execute programs on a JVM, we gain access to a parallel garbage collector, thus allowing multicore programming in a shared-memory setting.

Obviously, compiling OCaml sources to Java bytecode is not enough to provide interoperability between the two languages. It is already possible to call OCaml code from Java code thanks to the OCaml-Java project through various modalities:

- low-level mechanisms: callbacks (that is runtime registration of bare values and functions), and scripting (that is an engine for the `javax.script` package);
- high-level mechanism: wrapping of OCaml libraries by the `ocamlwrap` tool.

The `ocamlwrap` tool, presented in [7], allows the developer to wrap the various kind of OCaml values into Java classes while preserving the strong typing of the OCaml code. The tool is based on the generation of Java source files from OCaml compiled interface files.

In this article, we are interested in calling Java code from OCaml code. Unlike the approach used by the `ocamlwrap` tool, this is done by extending the OCaml type system to make it aware of Java types. In [6], we already presented such an extension, but it used *ad hoc* design and implementation. The new design is based upon an encoding of Java types into OCaml types, providing three major benefits:

- the implementation is less invasive and easier to maintain;
- the implementation is based on the well-tested original OCaml typer;
- the general scheme can be reused to interface other languages.

The encoding of Java types into OCaml types is based on the combined use of phantom typing, subtyping, polymorphic variants, and value-directed typing. The crucial features of the Java languages used in the encoding process are the ability to inspect the

class hierarchy at compile-time, and the fact that typing is nominal. The system as a whole could arguably be seen as a very flexible FFI[1], but we underline that, contrary to most FFI systems, it guarantees type safety and features a limited form of type inference.

In this paper, we first present the challenges that make the combination of OCaml and Java difficult in Section 2, and expose the key points of our system in Section 3. Then, Sections 4 to 8 provide the details of our design and implementation: actual encoding of Java type in Section 4, modifications to the OCaml typer in Section 5, code generation in section 6, and support for arrays and exceptions in respectively Sections 7 and 8. Finally, we compare our approach to related work in Section 9, and discuss possible evolutions in Section 10.

2. Challenges to interoperability

The type systems of the OCaml and Java languages are fundamentally different, and hence quite difficult to combine. OCaml is a multiparadigm language (supporting functional, imperative, and object-oriented programming) featuring a rich type system, while Java is an object language featuring a simpler type system.

Of course, it is more difficult to embed the typing of the richer type system into the simpler one. As we have shown in [7], it is not possible to translate all OCaml constructs to Java constructs by retaining complete type safety. However, a reasonable subset can be mapped and it is already quite useful in practice.

Embedding the simpler type system into the richer one is a tad easier, but it still comes with few challenges to overcome. In the remainder of this section, we list the main differences between the type system, particularly underlying the ones that make interoperability difficult.

Mostly versus Totally Statically Typed Although the Java language is mostly statically typed, it nevertheless features dynamic type tests. These runtime tests are arguably used to circumvent limits of the type system that make impossible to express complex properties over types. The OCaml language, on the opposite, is entirely statically typed, ensuring that no dynamic type check is ever required. Indeed, the compilation of OCaml is done by type erasure, meaning that (almost) no type information is retained at runtime.

The fact that OCaml is statically typed implies that we have to determine the type of the Java elements manipulated at compile-time. However, it is neither possible nor desirable to rule out the runtime tests of the Java language. For example, it is quite usual for Java libraries to store name-value bindings as a map from `String` instances to `Object` instances, and let the developer determine the actual type of the bound value by using either the `getClass()` method, or the `instanceof` operator.

Our embedding of the Java type system does obviously not modify the libraries already available for the Java language. As a consequence, we provide support runtime tests over Java instances. As the `ocamljava`-compiled programs run on a JVM, this is done with no effort. The developer only has to be aware of an hybrid model, where dynamic type tests are available for the Java instances, while the type of OCaml values cannot be retrieved at runtime.

Partial versus Complete Type Inference In Java, the developer is required to explicitly give the type of manipulated elements almost everywhere. In fact, only the *diamond* (*i.e.* `<>`) of generics provides support for a very limited form of type inference. As an example, it allows to write `Map<String, Object> s = new HashMap<>()` instead of the more verbose `Map<String, Object> s = new HashMap<String, Object>()`.

On the opposite, OCaml features full type inference, and the developer is almost never required to write any typing information in its program. In practice, type annotation are mainly used in OCaml when some subtyping is involved, or when the compiler would not be able to determine the type of a value (*e.g.* when deserializing a value).

It is unfortunately not possible to provide full type inference over Java types, as Java allows to overload names, and to decide which element to use through type-based decisions (possibly occurring at runtime). As an example, it is perfectly legal in Java to declare the following:

```
public class C {
  public int method() { ... }
  public int method(String s) { ... }
  public int method(Integer i) { ... }
  public int method(Object o) { ... }
}
```

Name overloading is possible as long as signatures are different. Indeed, while this notion of signature only comprises method parameters in the frame of the Java language, it also includes the return type in the frame of the JVM. Of course, name overloading is incompatible with full type inference, in particular for a language such as OCaml that uses a compilation scheme based on type erasure (thus ruling out any form of dynamic dispatch).

Our embedding of the Java type system does not provide full type inference over Java types, but still features a very limited form allowing the user to elide the types when there is no ambiguity. In the case of ambiguity, the compiler will issue an error message, and the developer will have to provide some type information to fix the error.

Nominal versus Structural Typing The Java object model is based on nominal typing, which means that the subtyping relationship between classes is essentially given by the class hierarchy that is explicitly built by referring to parent types (either classes or interfaces) through their names. This way of handling subtyping is not only simple, it also allows a developer to have maximum control over it. Alas, it prevents the use of a subtyping relationship not envisioned by the original developer. For example, the `String` class and `Collection` interface both define an `isEmpty()` method, but a developer cannot treat `String` and `Collection` instances in a uniform way, even if only using methods that are common to both.

On the contrary, the OCaml object model is based on structural typing, which means that the subtyping relationship is determined only by looking at the the signature of a class (*i.e.* the set of methods it defines), regardless of the class hierarchy. Listing 1 shows two class definitions, and a function calling some methods over a passed parameter. The type of the function, as inferred by the compiler is:

```
val f : < char_at : 'a -> 'b;
          start : unit -> 'c; .. > -> 'a -> 'b
```

and would still be legal (and the very same) without the class definitions. What it tells us is that `x` has to be instance that provides two methods. As can be seen from the use of type variable (`'a`, `'b`, and `'c`), the function is polymorphic. This example illustrates that structural typing is not interested in the actual class to be passed

[1] Foreign Function Interface

(that would be referred to by its name), but in the set of methods actually called.

```
1  class string = object
2    method char_at i = ...
3    ...
4  end
5
6  class thread = object
7    method start () = ...
8    ...
9  end
10
11 let f x i =
12   x#start ();
13   x#char_at i
```

Listing 1. Object use in OCaml.

Our embedding of the Java type system is based on the Java object model, and thus name-based. As we will see in Section 5, this will imposes to add some checks to the original OCaml inference engine, in order to guarantee that the inferred types are actual Java types. Without those tests, the OCaml inference engine may sometimes output types to be read as the conjunction of Java classes, for example requiring a given value to be at the same time of type `string` and `thread`.

Covariant versus Invariant Arrays In the Java language, arrays are covariant, which means that it is possible for example to pass a `String` array where an `Object` is expected. Most developers find this both intuitive and convenient, but there is a high price to pay for this flexibility. Indeed, each array store involves a dynamic type check to ensure that the element to be stored has a legal type, throwing an exception if not. This is necessary, as exemplified by the following code:

```
String[] strings = new String[] { ... };
...
Object[] objects = strings;
objects[idx] = new Integer(...);
```

At compilation time, the array store seems legal, as `Integer` is a subtype of `Object`. However, the array store will fail at runtime, as the `objects` reference actually points to an array of `String` instances, and can thus not be used to store an `Integer` instance.

As OCaml puts a strong emphasis on type safety, its arrays are invariant. This is less flexible but ensures that no type error will occur at runtime. As an aside, it would not be possible to switch to covariant arrays, as the compilation scheme is based on type erasure (the required type information may thus not be available at runtime).

Our embedding of the Java type system uses a mixed solution: arrays are treated as invariant, but the support for the cast operator of Java allows the developer to get some flexibility when needed, by giving up type safety. Section 7 gives more details on the way Java arrays are mapped to OCaml types.

3. Integration of the Java type system

In this section, we present a broad overview of the typer extension whose details are then given in Sections 4 to 8. We first explain the design decisions that led to the current implementation, and then expose its base elements. Finally, we present a simple example showing a practical use of the typer extension.

3.1 Design Decisions

Three key properties constrained the design space for our integration of the Java type system into the OCaml one:

- the typer extension shall not modify the existing syntax of the OCaml language;
- the typer extension shall produce easy-to-understand error messages;
- the typer extension shall allow compilation to *plain* and *efficient* Java bytecode.

The first property is important to be able to leverage the various tools from the OCaml ecosystem that work at the source level. Practically, this means that the developer is still able to use its usual editors, preprocessors, *etc.* Existing tools are not impacted by our typer extension. Moreover, the OCaml-Java compiler uses by default the original typer system, the extensions being explicitly activated by a command-line switch (namely `-java-extensions`).

The second property is important to provide the developer with a pleasant experience. In particular, if the developer makes a type mistake when manipulating Java instances, the error message should be as simple as for example "`java.lang.String is waited but java.lang.Thread is found`". The crucial element is that the developer should face error messages that immediately "make sense", without having to fully understand our typer extension as it would entail a steep learning curve.

The third property is important in order to promote OCaml-Java as an alternative language for the JVM. By *plain* and *efficient*, we mean that an operation such as a method call should be translated to a simple `INVOKEXYZ` Java instruction, and not go through a complex mechanism such as reflection or method handles. Similarly, accesses to instance fields or array elements should be mapped to simple instructions, and also allow the compiler to avoid value boxing when possible.

3.2 Base Elements

We present here the most prominent characteristic of our typer extension: representation of Java types, as well as generic mechanism used to create and manipulate Java instances.

Primitive Java types are simply mapped to predefined OCaml types. Non-primitive Java types are mapped to a dedicated OCaml abstract type, namely `'a java_instance`. Of course, this newly-introduced type will enjoy custom treatment to fit the Java model, and particularly subtyping between Java types. The type parameter `'a` is used to designate a particular Java type; for example, an instance of the Java `java.lang.String` class will be represented on the OCaml side by type `java'lang'Object java_instance`. The way to write the Java class name is changed from `java.lang.String` to `java'lang'String` in order to abide the lexical and syntactic rules of the OCaml language.

Once it is possible to designate Java types, it is necessary to devise means to create and manipulate Java instances. To this end, we propose a mechanism akin to the one used by the original OCaml implementation for the treatment of `printf`-like functions. To handle such functions, the typer analyzes at compile-time the format string in order to determine the actual types of the parameters that should be passed to the function. We use the same principle to create Java instances, leading to the following expression to create an `Object` instance: `Java.make "java.lang.Object()" ()`.

In addition to these *core* elements, we also provide two mechanisms to reduce code verbosity. The first mechanisms is the pos-

sibility to "import" Java packages through the OCaml `open` directive, and then to refer to classes by their simple names rather than by their fully-qualified names, thus leading to the following code for the previous example: `Java.make "Object()" ()`[2]. The second mechanism is the possibility to replace parameters types by a simple "_" (underscore) as long as there is no ambiguity. As an example, it allows to simply write:

```
Java.make "Thread(_,_,_)"
```

rather than:

```
Java.make "Thread(ThreadGroup,Runnable,String)"
```

3.3 Example: Minimal Swing Application

In order to give a taste of practical uses of the typer extension, we present in this section the OCaml code needed to to build a Swing GUI. We will see how instances are created, how methods are called, and how interfaces are implemented.

Listing 2 shows build a one-button Swing frame can be created from several parameters (respectively title, width, height, button label, and action associated with button click). Line 2 opens the `Java` module, allowing to use its `make` and `call` functions unqualified. Lines 3 and 4 respectively create the frame and its button: the `make` function receives as its first parameter the signature of the Java constructor to be called. Lines 5 to 8 call various methods to build the frame: the `call` function receives as its first parameter the signature of the Java method to be called. As long as there is no ambiguity on the designated method, parameter types can be replaced by underscores acting as wildcards.

```
1  let make_frame ttl w h lbl act =
2    let open Java in
3    let f = make "JFrame(String)" ttl in
4    let b = make "JButton(String)" lbl in
5    let p = call "JFrame.getContentPane()" f in
6    call "JFrame.setSize(_,_)" f w h;
7    call "JButton.addAciontListener(_)" b act;
8    call "JFrame.add(Component)" p b;
9    f
```

Listing 2. Creating a Swing Frame in OCaml.

The type of the `make_frame` function, as inferred by the compiler is the following:

```
val make_frame :
  java'lang'String java_instance ->
  int32 ->
  int32 ->
  java'lang'String java_instance ->
  java'awt'event'ActionListener java_instance ->
  javax'swing'JFrame java_instance
```

The `int32` type is a plain OCaml one, and the `pack'Class java_instance` type is the OCaml type for instances of the Java class whose fully-qualified name is `pack.Class`. Instances of Java strings can be easily converted from/to OCaml strings through the `JavaString` module. The `ActionListener` instance is a bit different, as it designate an interface that can hence not be directly instantiated.

It is possible to implement Java interfaces through *proxies*. The developer has to indicate which interface she wants to implement, along with an OCaml object actually providing the implementation for the methods of the interface. Listing 3 shows how an `ActionListener` can be implemented. The interface consists in one single method, namely `ActionPerformed`, taking a parameter of type `ActionEvent`. As the parameter is not used by the method implementation, it remains unnamed. The `quit` value has type `java'awt'event'ActionListener java_instance` and can thus be passed to the `make_frame` function.

```
1  let quit =
2    Java.proxy "java.awt.event.ActionListener"
3      (object
4        method actionPerformed _ =
5          exit 0
6      end)
```

Listing 3. Implementing a Java interface in OCaml.

4. Encoding of Java types into OCaml types

The encoding of Java types into Java types is based on the combination of several techniques already available to OCaml developers, precisely: phantom types, polymorphic variants and value-directed typing. We begin this section by introducing these techniques, and then move on to their use inside the OCaml-Java project.

4.1 Phantom types

A phantom type is a parametrized type whose some parameters only appear and its declaration and not in its definition. For example, the following record type:

```
type ('a, 'b) resource = { value : 'a }
```

is a phantom type as the type parameter `'b` only appear on the left-hand side. Phantom types are commonly used to encode additional properties into types, with the benefit that they incur no runtime overhead.

A classical use is to implement access rights to resources. Consider a program where some resources have read-only rights while other have read-write rights. By embedding the rights into the type system, one can rely on the compiler to ensure that all access conform to the rights.

Listing 4 shows a possible encoding of such a system. The two permissions are translated into abstract types, and two functions are provided to create resources with given rights. Then, the `read` function is made available to all resources by accepting any kind of resource through the `'b` parameter of the `('a, 'b) resource` type, while the `write` function is only available to read-write resources by explicitly requesting `read_write` as the second component.

```
1  type ('a, 'b) resource = { value : 'a }
2  type read_only
3  type read_write
4  val make_ro : 'a -> ('a, read_only) resource
5  val make_rw : 'a -> ('a, read_write) resource
6  val read : ('a, 'b) resource -> 'a
7  val write :
8     ('a, read_write) resource -> 'a -> unit
```

Listing 4. Phantom types used to model access rights.

Other classical uses of phantom types include encoding of units of measurement (*e.g.* whether a given value is expressed in meters or feet), some state (*e.g.* whether a socket is connected), or some generic property over a data structure (*e.g.* whether a given list can be empty).

[2] As in Java, the `java.lang` package is always opened.

4.2 Polymorphic variants

Polymorphic variants are a flexible alternative to *classical* variants or sum types. Unlike *classical* variants, their constructors do not have to be unique and are also not tied to a given module. It follows that a given constructor is not tied to a given type. This does not only allows reuse of constructors in disjoint situations, but also to define a sum type as a subset of another one. For example, to model a generic notion of flags, and a particular version of flags for classes, one can write in OCaml:

```
type flags = [ ‘Public; ‘Private; ‘Synchronized ]
type class_flags = [ ‘Public; ‘Private ]
```

where the backtick is used to distinguish constructors of polymorphic variants from constructors of *classical* variants.

The ability to define a polymorphic variant as the subset of another one allows to enforce data structure invariants. In our example, when defining a class, the type `class_flags` will be used to be sure that one does not try to mark a class with the `‘Synchronized` flag. Moreover, it is possible to define a function working over the `flags` types, and to pass it values of the `class_flags`. Thus achieving at the same time genericity and safety, without resorting to conversion between distinct sum types.

OCaml developers frequently mix polymorphic variants with phantom types. Not only because they save from the need to define additional types (thus polluting the namespace), but also because they carry a notion of subtyping. Indeed, our resource example can be rewritten to use polymorphic variants, as shown by Listing 5. The crucial element is the use of the `[> ...]` form that, contrary to the `[ ... ]` form, states that the expected type should contain *at least* the constructors between the bracket (and thus may contain additional constructors).

```
1  type (’a, ’b) resource = { value : ’a }
2  val make_ro : ’a -> (’a, [ ‘Read ]) resource
3  val make_rw :
4     ’a -> (’a, [ ‘Read | ‘Write ]) resource
5  val read : (’a, [> ‘Read]) resource -> ’a
6  val write :
7     (’a, [> ‘Write]) resource -> ’a -> unit
```

Listing 5. Phantom types with polymorphic variants.

In our contrived example, the two versions are not very different, but it is mainly due to the fact that we only have two levels for the resource permissions; it is then easy to just use polymorphism to access both levels as done in Listing 4. However, if we define additional permissions with operations that are only available to given combinations of permissions, the first proposed encoding will become tedious. On the opposite, the second encoding can be trivially extended to additional permissions, relying on subtyping (*i.e.* the fact that `[ ‘X ]` is a subtype of `[ ‘X ; ‘Y ]`) and open polymorphic variants (*i.e.* the `[> ‘Z ]` form).

Moreover, the OCaml language allows to finely control the subtyping relationship between types through variance annotations over type parameters. A type parameter can be declared with a `+` in order to indicate that it is covariant, or a `-` to indicate that it is contravariant. The default, that is the absence of any sign annotation, is to consider the type parameter as invariant. Suppose that we have two types `t1` and `t2` such that `t1` is a subtype of `t2`. If we declare `type +’a plus`, then `t1 plus` is a subtype of `t2 plus`. On the opposite, if we declare `type -’a minus`, then `t2 minus` is a subtype of `t1 minus`. In OCaml, due to the coexistence of type inference and subtyping, it is often necessary for the developer to use explicit coercions (whose notation is `(expr :> type)`) to comply with the typing rules. However, it should be noted that coercions are always checked at compile-time and thus type-safe.

Java type	OCaml type	notes
`boolean`	`bool`	
`byte`	`int`	*63-bit integer*
`char`	`int`	*63-bit integer*
`double`	`float`	*double precision*
`float`	`float`	*double precision*
`int`	`int32`	
`long`	`int64`	
`short`	`int`	*63-bit integer*
`pack.Class`	`-’a java_instance`	

Table 1. Mapping of Java types.

4.3 Value-directed typing

The original OCaml standard library ships with a module named `Printf`, which unsurprisingly provides functions allowing `printf`-like formatting. Such functions take as their first parameter a format string that can contain formatting instructions introduced by the `%` character. For example, the following allows to print a string and an integer as the values of a key and its binding:

```
Printf.printf "key: %s binding:%d\n" x y
```

However, the type of this function cannot be expressed in *vanilla* OCaml as its actual type depends on the contents of the format string. For this very reason, there is a special typing rule in the compiler to handle format strings, resulting in what we call *value-directed* typing to underline that the type of an expression depends on a string literal.

Now examine how `printf` is handled at the typing level. If we ask toplevel to output the type of the function, we get:

```
(’a, out_channel, unit) format -> ’a
```

and if we ask it the type of the expression “`printf "int: %d"`”, we get:

```
int -> unit
```

Wherever the `format` type is expected, the OCaml compiler also accepts a literal string as a valid expression. The aforementioned literal string is then parsed to determine the list of parameters that should actually be passed to `printf` to make it able to properly render the format string. Then, these parameters are bound to the `’a` type parameter of the `format` type, so that the compiler can treat the other parameters by applying the usual typing rules.

4.4 Combination in OCaml-Java

Encoding of the class hierarchy Table 1 summarizes the mapping from Java types to OCaml ones. Primitive Java types are just mapped to OCaml predefined types. Then, the most interesting type is `java_instance` that accounts for Java reference types (besides arrays whose handling is detailed at Section 7). The abstract `java_instance` type accepts a type parameter that is used to precisely designate the represented Java type.

The encoding of the Java type is done by combining phantom typing and polymorphic variants. The Java type `Object` is simply represented by the OCaml type:

```
[ ‘java’lang’Object ] java_instance
```

Type	Meaning and example
java_constructor	constructor signature "java.lang.Object()"
java_method	method signature "java.lang.Object.wait():void"
java_field_get	field signature "java.lang.Thread.MAX_PRIORITY:int"
java_field_set	field signature "java.lang.Thread.MAX_PRIORITY:int"
java_type	class, interface, or array type "java.lang.String"
java_proxy	interface type "java.lang.Comparable"

Table 2. Format types.

Similarly, the Java type `Thread` is represented by the OCaml type:

```
[ ‘java’lang’Object
| ‘java’lang’Runnable
| ‘java’lang’Thread ] java_instance
```

thus ensuring that the type for threads is actually a subtype of the type for objects.

More generally, a Java reference type is encoded as a `[ c0 | ... | cn ] java_instance` OCaml type, where the `ci` constructors represent all the parent types of the type to be encoded. These parents include all the super classes, as well as all the implemented interfaces. Conceptually, one can think of these constructors as representing "*all the reference types the designated instance can be **safely** casted to*". It is noteworthy that the type parameter of the `java_instance` type is declared contravariant, meaning that `[ c0 | ... | cn ] java_instance` is a subtype of `[ d0 | ... | dp ] java_instance` iff the `ci` constructors form a subset of the `dj` constructors, which is coherent with the Java semantics.

This encoding is based only on two properties of the Java language: (i) the typing of reference types is nominal, and (ii) the class hierarchy is fully determined at compile-time. It is interesting to notice how the encoding differs from the ones exposed in [8]. In [8], the authors show *how* phantom types can be used in Standard ML in order to encode a subtyping relationship, while our encoding *uses* the combination of subtyping and phantom typing to encode the Java class hierarchy into OCaml types.

Manipulating instances Now that we are able to designate Java types, it is necessary to devise means to create instance, access their fields, invoke their methods, *etc*. All these manipulations are based on variation over the *value-directed* typing presented above. We introduction a `Java` module providing the following functions:

```
make : ’a java_constructor -> ’a
call : ’a java_method -> ’a
get : ’a java_field_get -> ’a
set : ’a java_field_set -> ’a
instanceof : ’a java_type -> ’b java_instance -> bool
cast : ’a java_type -> ’b java_instance -> ’a
proxy : ’a java_proxy -> ’a
```

where all the types appearing in the first parameters of the various functions are akin to the `format` type used by the `printf` function. Table 2 presents the semantics, and example strings for the various types.

Likewise to what is done for the *printf* function, the `’a` parameter is used to encode the actual types of the call, just using the type mapping presented above. For example, the type of "`make "java.lang.Integer(int)"`" is:

```
int32 -> [ ‘java’lang’Object | ‘java’lang’Number
  | ‘java’lang’Integer | ‘java’io’Serializable
  | ‘java’lang’Comparable ] java_instance
```

However, method parameters use the open form of variants (*i.e.* `[> ...]`), so that for example it is possible to to passe an `Integer` instance where an `Object` is waited. Without this slight modification, the developer would be required to explicitly coerce from `Integer` to `Object` in such situations. As a result, the type of "`call "java.lang.Object.wait():void"`" is:

```
[> ‘java’lang’Object ] java_instance -> unit
```

Adding sugar Thus far, we are able to designate Java types, and to manipulate Java instances. However, the literal strings used to refer to Java elements are pretty verbose. We thus introduce two elements to reduce verbosity:

- an equivalent to the Java `import` directive;
- a wildcard to be used instead of actual types.

The original OCaml provides a mechanism used to open a module, that allows to access its elements without prefixing them with the module name. We modify the treatment of the `open` directive, so that "`open Package’pack`" is the OCaml equivalent to the "`import pack.*`" Java directive. As a result, the constructor of the `Object` class can be called by simply writing `make "Object()"`.

As previously seen, it is also possible to replace any type by the underscore character (*i.e.* `_`). The lookup mechanism will accept any type where an underscore is used. If the lookup mechanism returns one element, it is simply used. If the lookup mechanism returns more than one element, then the compiler outputs an error message with the choices leading to the ambiguity. While its implementation is much more simple, this wildcard mechanism can be seen as a very limited form of type inference, allowing the user to both write shorter code, and request the compiler to determine the type of a partially-specified element.

5. Amendments to the Original OCaml Typer

Up to this point, we left the original OCaml typer almost untouched, only adding support for the format strings used to designate the various Java elements (constructors, methods, fields, *etc.*), as well as support for the enhanced `open` directive. However, as we will see in this section, we felt the need to slightly modify the typer for usability reasons (as opposed to correctness reasons).

5.1 Shortening Java types

The most prominent problem regarding usability is that Java types as encoded as phantom polymorphic variants are quite verbose. The developer should be able to specify a Java type without having to write down its complete hierarchy, which can be lengthy. Of course, the OCaml language being based on type inference, the type of manipulated entities is very rarely expressed by the developer; however, the types of exported values appear in `mli` files (*i.e.* top-level module signatures).

We thus provide a convenient shorthand notation for Java types that relies on the principle that we have two representation for such types: a core representation and a surface notation. The former has been the one exposed at Section 4.4, while the latter has been

introduced at Section 3.2. As a consequence the type of threads is simply `java'lang'Thread java_instance`, rather than:

```
[ javalangObject
| javalangRunnable
| javalangThread ] java_instance
```

A pre-treatment transforms the surface notation into the the core representation by looking up for the designated class in the classpath. Once found, we simply recursively determine all its parents to build the set of constructors of the polymorphic variant.

Symmetrically, a post-treatment transforms the core representation into the surface notation, such that types output by the compiler (*e.g.* in error messages) use the lightweight notation rather than the internal one. This post-treatment is trivial: we iterate over the constructors and remove all constructors that designate the parent of another constructor. At the end of the process, only one constructor remains: the one that is not a parent, and hence represent the Java class encoded by the set of constructors.

It is also possible in the surface notation to take advantage of opened package to elide the package from the class name, replacing it by an underscore. Thus leading to the shortest notation for the type of threads: `_'Thread java_instance`.

Finally, besides `java_instance`, we provide another type in the surface notation, namely `java_extends`. The two differ only in the openness of the polymorphic variant in the core representation. Precisely:

- `pack'Class java_instance` is translated to `[ c0 | ... | cn ] java_instance`;
- `pack'Class java_extends` is translated to `[> c0 | ... | cn ] java_instance`.

5.2 Enforcing Java types

There are great incentives in encoding the typing of Java elements into existing OCaml types, uniformity and maintainability coming first to mind. However, this also comes with an obvious drawback: the typing rules of OCaml are then applied to Java elements, possibly leading to unexpected types.

Indeed, the unification algorithm of OCaml when applied to our encoding of Java types can output types that are perfectly legitimate OCaml types but have no sensible equivalent in Java. Consider for example the code sample of Listing 6, where the developer arguably made a typo, using `x` instead of `s` at line 4. The type, as inferred by the compiler, is shown by lines 6-12 using the core representation.

```
1  let f x =
2    let open Java in
3    let s = call "Integer.toString()" inst in
4    call "String.charAt(_)" x 0
5
6  val f : [> `java'lang'Object
7    | `java'lang'Number
8    | `java'lang'Integer
9    | `java'io'Serializable
10   | `java'lang'Comparable
11   | `java'lang'CharSequence
12   | `java'lang'String ] java_instance -> int
```

Listing 6. Inferring *impossible* types.

The type of `f` makes perfect sense in the structural typing of OCaml. It simply indicates that the type of `x` is the conjunction of the type of `Integer` and `String`. While perfectly correct with respect to the code, it suffers from a major problem: it is not possible to build a Java instance that matches this type.

To avoid such types, we amend the OCaml compiler in order to check the output of the unification function in order to ensure that all inferred Java types are indeed *possible* Java types. Checking whether a given set of constructors represent a *possible* Java type is obvious using the post-treatment described above to output Java types. If the post-treatment ends up with only one constructor, then the set represents a simple class. Otherwise, the presence of several constructors represents a conjunction of several classes.

In some sense, this mechanism can be considered as a *fail-fast* measure. Indeed, it informs the developer as soon as possible that she is combining types in a way that makes no sense in the Java typing system. Accepting the *impossible* Java type would as a matter of fact entails no unsafety, as the developer would in practice be unable to build an instance to be passed to the `f` function.

6. Code Generation

Once the typer of the compiler has checked that the passed source abide to the typing rules, subsequent compiler phases are responsible for actual code generation. We describe in this section how the *special* functions used to call Java constructors, methods, to access fields are compiled into Java bytecode.

As seen in the previous section, the compiler will issue an error if the developer writes a constructor (or method or field, for that matter) signature that is ambiguous. Similarly, an error will also be issued if there is no match for a given signature. In practice, this means that the element to called or accessed is completely determined at compile-time. This allows to drop the string literal used to encode the signature. It is noteworthy that this is different from the compilation of `printf`-like functions that need the format string both at compile-time (in order to determine the number and types of expected arguments) and at runtime (in order to render the string to output).

The referenced element being totally determined at compile-time, the compiler is able to output efficient bytecode, avoiding to resort a dynamic mechanism such as reflection. For example, the expression "`Java.make "Object()" ()`" is translated in the following sequence:

```
new java.lang.Object
dup
invokespecial java.lang.Object.<init>()
```

which is the very same code that is generated by the `javac` compiler.

However, Java instances are represented in OCaml as a *custom* type. An OCaml *custom* type is an abstract type that can be provided with specific functions for comparison, hashing, and serialization. The Java instance is boxed in a *custom* value. Nevertheless, the overhead entailed by this extra indirection is mitigated by the unboxing optimization performed by the compiler, and is also reinforced by code inlining.

Recent versions of the OCaml-Java compiler do a very aggressive unboxing. For example, unboxed values can be used as parameter or return values to/from functions. In practice, most of the time, the Java instances will be boxed only when stored in OCaml data structures. Performance of Java instance manipulations is hence reasonably close to the equivalent manipulations directly done us-

ing the Java language.

There is only one function that cannot be directly mapped to plain Java bytecode: the `proxy` function. The function takes as its first parameter a literal string giving the name of an interface, and as its second parameter an OCaml object implementing the methods specified in the interface. At compile-time, as for other functions, the string literal is used to determine the actual type of the OCaml object and then discarded.

At runtime, a Java instance is built using the method named `Proxy.newProxyInstance`. This instance is responsible for receiving method calls, and dispatching them to the OCaml method. The Java instance is also responsible for converting parameter and return values back and forth.

7. Handling of Arrays

In this section, we examine how array types are embedded into our encoding. This is indeed an interesting problem, as we want to ensure at the same time genericity and decent performance. This leads to a custom encoding which leverage advanced features of the OCaml type system to provide genericity. It is important to notice that Java itself does not support genericity for arrays types. It can be observed by looking at the `java.util.Arrays` class that provides for example distinct methods to sort arrays of type `byte[]`, `char[]`, *etc.*

7.1 The Case for Specialized Arrays

In OCaml, arrays are represented through the `'a array` type; however, their specific runtime representation makes impossible to use this type to represent Java arrays. Reusing this type would in fact entail a systematic copy of the data from the Java array to its OCaml counterpart. This would not only result in a significant overhead, but would also makes it quite difficult to share an array instance between OCaml and Java code, which may be needed by some Java libraries.

Then, another possibility would be to define a dedicated `'a java_array` type that would provide a custom representation to carry actual Java array instances, leading to a module akin to the one showed by Listing 7. It is noteworthy that the module does not provide any means to *directly* create an `'a java_array` because in this encoding we would like to restrict the `'a` parameter to Java types only, ruling out OCaml types.

```
1 type 'a java_array
2 val length : 'a java_array -> int32
3 val get : 'a java_array -> int32 -> 'a
4 val set : 'a java_array -> int32 -> 'a -> unit
```

Listing 7. Uniform representation for Java arrays in OCaml.

Anyway, this representation is not satisfactory, as it forces to have a common representation not only for the various `'a java_array` instances, but also for the various `'a` instances. While the former is not a major problem (it would still incur a dynamic check to query the actual type of the array before any get/set operation), the latter would imply to always use a boxed representation for array elements. This boxing is not a concern for arrays of references but is a huge penalty in the case of primitive elements.

As a consequence, in order to reach decent performance, we can provide different implementations for the various array types, as shown by Listing 8. Then, we face another problem, the genericity one. It could be solved by adding another type to the modules to represent the type of the elements, but we statically know the type of elements only for primitive arrays. We thus decided to define all array types with a type parameter, and ensure that the functions responsible for array creations will enforce the correct value for the type parameter. Practically, it means that the equivalent of Java type `int[]` is `int32 java_int_array`, as shown by Listing 9.

```
1  module type IntArray = sig
2    type t
3    val length : t -> int32
4    val get : t -> int32 -> int32
5    val set : t -> int32 -> int32 -> unit
6  end
7
8  ...
9
10 module type LongArray = sig
11   type t
12   val length : t -> int32
13   val get : t -> int32 -> int64
14   val set : t -> int32 -> int64 -> unit
15 end
16
17 ...
```

Listing 8. Possible specialized representation for Java arrays in OCaml.

```
1  module type IntArray = sig
2    type e = int32
3    type 'a t
4    val length : e t -> int32
5    val get : e t -> int32 -> e
6    val set : e t -> int32 -> e -> unit
7  end
8
9  ...
10
11 module type ReferenceArray = sig
12   type 'a t
13   val length : 'a t -> int32
14   val get : 'a t -> int32 -> 'a
15   val set : 'a t -> int32 -> 'a -> unit
16 end
```

Listing 9. Actual representation for Java arrays in OCaml.

7.2 The Reach for Genericity

Through functors The representation exposed by Listing 9 is particularly interesting as the various specialized array types actually share the same signature at the module level, precisely:

```
module type ArraySignature = sig
  type e
  type 'a t
  val length : e t -> int32
  val get : e t -> int32 -> e
  val set : e t -> int32 -> e -> unit
end
```

It is hence possible to define functors over that signature to write functions that operate on any kind of array. For example, a generic `iter` routine may be written:

```
module Iterator (A : ArraySignature) = struct
  let iter f a =
    let i = ref 0l in
    let l = A.length a in
    while !i < l do
```

```
      f (A.get a !i)
      i := Int32.succ !i
    done
end
```

The `Iterator` functor can then be applied to any specialized array module in order to get an `iter` function for a particular kind of arrays.

***Through* GADTs** However, functors are quite heavy to manipulate as one will need to write a new functor to provide new generic operations. For this reason, we provide another abstraction over the various kinds of arrays to unify them into a single type. This single type will act as a wrapper around the various array types. This is done through a GADT with three type parameters whose semantics is:

1. the type of array elements;
2. the type of array indexes;
3. the type of wrapped array.

Listing 10 shows the resulting module declaration.

```
module type JavaArray = sig
  type (_, _, _) t
  val wrap_int_array : int32 IntArray.t ->
    (int32, int32, int32 IntArray.t) t
  val wrap_long_array : int64 LongArray.t ->
    (int64, int32, int64 LongArray.t) t
  val wrap_reference_array :
    'a ReferenceArray.t ->
    ('a, int32, 'a ReferenceArray.t) t
  ...
  val length : (_, _, _) t -> int32
  val get : ('e, 'i, _) t -> 'i -> 'e
  val set : ('e, 'i, _) t -> 'i -> 'e -> unit
  val wrapped : (_, _, 'r) t -> 'r
end
```

Listing 10. Actual representation for Java arrays in OCaml.

Once an array is wrapped, the usual `length`, `get` and `set` function can be called in a generic but type-safe way, thus allowing to write generic code over the `JavaArray.t` type. For example, the aforementioned `iter` routine may be written:

```
let iter f a =
  let i = ref 0l in
  let l = JavaArray.length a in
  while !i < l do
    f (JavaArray.get a !i)
    i := Int32.succ !i
  done
```

Moreover, the power of GADTs also allows us to encode 2D arrays in the very same type, by just adding the following functions (to wrap 2D arrays, and query the length of a sub array):

```
val wrap_int_array2:
  int32 IntArray.t ReferenceArray.t ->
  (int32,
   int32 * int32,
   int32 IntArray.t ReferenceArray.t) t
...
val length_sub: (_, int32 * int32, _) t ->
  int32 -> int32
```

No change is needed to the `get` and `set` functions. The type system ensures that whenever an element of a 2D array is accessed, the passed index is a couple of integers rather than a simple integer.

Summary The proposed way of implementing arrays provides an acceptable compromise: we grant both efficiency and genericity at the price of several module declarations. The key property of this implementation is that the developer will pay a performance penalty only when actually taking advantage of genericity: polymorphic code will go through an indirection while monomorphic code will be as efficient as possible. It is also noteworthy that the kind of genericity made possible is greater than what is available in the Java language. Finally, this scheme based on specialized implementations tied into a common one through a GADT is generic and can be used in a variety of situations.

8. Handling of Exceptions

The OCaml and Java languages both support exceptions, with similar semantics. Precisely, the only difference is that OCaml exception handlers can enclose any expression while Java handlers can only enclose sequences of instructions. This lead to slightly different implementations (in OCaml the values are popped from the stack until the appropriate handler is found, while in Java the stack is outright emptied), but the code generator of the OCaml-Java compiler performs the necessary transformation so that the semantics are aligned.

In OCaml, the type of exceptions (namely `exn`) is basically a sum type with the extra property that it is open. The openness means that, contrary to other sum types, there is not a single place where all constructors are declared; an `exception E` directive adds a constructor `E` to the type. The constructor can, as ordinary sum types, also declare some attached types trough a declaration of the following form: `exception E of t0 * ... * tn`.

In Java, exceptions are classes that inherit (directly or indirectly) from the `java.lang.Throwable` class. In both languages, when a exception handler is declared, it is possible to indicate to which kind of exception it applies: through a constructor name in OCaml, and through a class name in Java.

To be able to catch Java exceptions in OCaml code, which is necessary as soon as it is possible to call Java methods, we have declare a new constructor through `exception Java_exception of ...`. This allows to catch Java exceptions just the same way we catch ordinary OCaml exception. The remaining question is the one about the type(s) to attach to this newly-introduced constructor.

As Java exceptions inherit from `java.lang.Throwable`, the obvious answer would be to have the type `java'lang'Throwable java_extends`. Unfortunately, such a type would be refused by the OCaml typer: `java_extends` uses the open form of polymorphic variants (*i.e.* `[> ...]`), which in turn implies the use of an implicit type variable `'a` that is used to track possible additional constructors. However, it is not possible to declare an exception carrying a type variable: this would result in polymorphic values to be attached to an exception instance, and would break type safety.

We thus declare the following exception constructor: `Java_exception of java'lang'Throwable java_instance`. Then, the developer will have to resort to `Java.instanceof` to discriminate the different exceptions, as it is not possible in OCaml to match over a type. Thus leading to code handlers like the ones in Listing 11.

```
open Package'java'io

try
  ...
with
| Not_found ->
```

```
    (* usual OCaml exception *)
    ...
| Java_exception e
    when Java.instanceof "IOException" e ->
    (* any Java exception inheriting from
       java.io.IOException *)
| Java_exception _ ->
    (* any other Java exception *)
    ...
```

Listing 11. Handlers mixing OCaml and Java exceptions.

9. Related Work

In this section, we review alternative systems providing language interoperability, whether they are directly integrated into a language or proposed as external tools. The original OCaml implementation [12] supports a limited form of interoperability with the C language through two mechanisms: *callbacks* and *externals*. Callbacks allow the developer to register some value on the OCaml side that can then be accessed on the C side. As such values can be functions, this allows to call OCaml code from C code; however, the developer is in charge of converting values between the type systems. Externals allow the developer to indicate that a given function will be implemented in C. The type of the function has to be explicitly given, and the developer is in charge of converting values between the type systems.

On top of these low-level foundations, several project have proposed easiest way to interact with C libraries. The `camlidl` project [10], as its name suggests, relies on an IDL (Interface Description Language) to describe a set of C functions and how their types map to OCaml types. From this description, the tool generates the necessary OCaml and C files with the boilerplate code responsible for type conversions. More recently, the `ctype` [15] project relies on a combinator-based OCaml library that allows to represent at runtime the type of C functions, and then to invoke them. Both projects are essentially interested in making easier to call C from OCaml, providing very limited support for the other direction. They succeed in their endeavor, greatly simplifying OCaml developments involving C code. However, they are not type safe, and an error in the description of a function to be interfaced with results in a runtime error.

It is also possible to interface the original implementation of OCaml with Java through the `camljava` project [11] that is based on the JNI framework. This provides a low-level access to Java elements, but can be combined to O'Jacaré [4] to obtain a higher-level interface. O'Jacaré uses an IDL to let the developer describe the set of classes to generate bindings for. The generated bindings present the developer with OCaml classes that are the counterpart of the Java ones. The problem of Java overloading is circumvented by allowing the developer to rename mapped elements.

Following the same principles than O'Jacaré, versions $1.x$ of the OCaml-Java project were based on a tool, namely Nickel [5], generating OCaml definitions and Java stubs from a mapping file written in XML. While we first considered polymorphic variants to overcome the overloading problem, we finally settled on name mangling because of the extra overhead entailed by the use of polymorphic variants. Listing 12 compares the two encoding for the case of the `wait` methods from the `Object` class.

```
class type java'lang'Object = object
  method wait : [ `LongInt of int64 * int32
                | `Unit ] -> unit
  ...
end

class type java'lang'Object = object
  method wait : unit -> unit
  method wait2 : int64 -> int32 -> unit
  ...
end
```

Listing 12. Overloading of Java methods in Nickel.

The OCamIL [3] project, whose compiler produces MSIL to be run on a .NET virtual machine, also used an adaptation of O'Jacaré tailored to the .NET object model. While this approach "just works" in practice, it entails several problems. First, it somewhat complexifies the build process that has to accommodate a code-generation stage. More importantly, errors messages are not always easy to decipher, mainly due to the mixing of the two objects models. For example, as OCaml classes are used to represent Java classes, one can pass any object implementing the same methods as the Java classes. This is accepted by the OCaml compiler because of its structural typing, but will fail at runtime as the instance should be an OCaml object holding an instance of a Java object.

Interestingly enough, earlier attempts to port a functional language to a *managed* runtime were actually based on typer extensions. For example, MLj [1] and SML.NET [2] ported the SML language to respectively the JVM and the .NET platforms by extending the type system of SML to make it aware of the object model of the underlying platform. This is probably due to the fact that the SML language, unlike OCaml, does not possess support for object-oriented programming; it was thus natural to add such a layer. While our typer extension is much more modest as it does not provide a complete object layer but simply encodes it into existing OCaml constructs, the practical result is quite comparable. In MLj, as in OCaml-Java, it is possible to create new instances, access their fields, call their methods, and also to implement interfaces.

More recently, the Scala language [13] has been designed with interoperability in mind. Its type system has hence been devised to be able to easily access to Java elements, but also to be able to produce classes that can be directly used by Java developers with no further processing. Scala also provides support for object-oriented and functional programming, presenting the developer with a uniform system. The use of Java elements from Scala is fully transparent for the developer, and full support for generics is provided. The object system used by Scala is basically the one of Java, thus avoiding impedance mismatch.

The Clojure language [9], a revival of the LISP language specifically designed to be hosted by a JVM, also provides a tight coupling with the Java type system. It allows to create and manipulate Java instances just as Clojure entities. Likewise to our `Java.proxy` function, Clojure provides means to dynamically implement a Java interface. Moreover, it can save a script as a class file, so that it can be then used from Java code. The only element of the Java type system not supported by Clojure is generics. However, this comes as no surprise, as the Clojure language is a dynamic language (thus interfacing with Java at runtime) and Java compiles generics by type erasure (thus retaining no type information at runtime).

The F# [14], that targets the .NET platform, and has been designed to easily interoperate with the C# language, is quite similar to OCaml. It could be argued that it was in beginning basically "*OCaml with a totally different object system*", the replacing object system being the one behind C# (or more generally behind the CLR of the .NET platform). Since then, the F# language evolved and is

now more distant from OCaml than it was initially. Manipulating C# entities from F# sources is transparent for the developer and the ML polymorphism of F# directly maps to C# generics, thus providing a consistent developer experience.

Finally, besides full-blown language implementations, some systems are indeed only concerned with the interoperability between programming languages. For example, the well-known SWIG project [16] allows to generate wrappers to C[++] library for a large range of languages. The tool is arguably based on an IDL, but its format is so close to C header files that in practice, it is often possible to actually use the C header files with no modification. From a developer standpoint, the SWIG approach suffers from the same problems raised by the Nickel or O'Jacaré projects. Moreover, as for these projects, the generated wrappers entail some overhead that is most of the time avoided when the interoperability layer is actually built into the language itself. When a tight integration between the language is not necessary, and when calls from one language to the other involve lengthy computations, it is also possible to resort to solutions originally designed for distributed systems. For example, the Thrift framework [17] allows to connect code written in various languages in a client-server mode. The Thrift tool generates code to exchange data in binary form between the languages from a file describing the various services that can be called.

10. Future Work

10.1 Generics

The current implementation provides full support for Java raw types, thus just ignoring generics where present. As a consequence, the first task will be to enhance the implementation to support generics. Fortunately, polymorphic variants, like plain sum types, can see their constructors carry values. The types of these values can contain type variables (*e.g.* `'C of 'a`), and can hence be used to represent the generic parameters. We can then map the Java type `Collection<E>` to the OCaml type:

```
[ 'java'lang'Object
| 'java'lang'Iterable of 'e
| 'java'lang'Collection of 'e ] java_instance
```

The types inevitably become quite verbose when instantiated, leading to the following one for a collection of threads:

```
[ 'java'lang'Object
| 'java'lang'Iterable
  of ([ 'java'lang'Object
      | 'java'lang'Runnable
      | 'java'lang'Thread ] java_instance)
| 'java'lang'Collection
  of ([ 'java'lang'Object
      | 'java'lang'Runnable
      | 'java'lang'Thread ] java_instance) ]
java_instance
```

The main problem, however, is not related to verbosity but to the fact that type expressions are significantly more difficult to produce (and thus, also, to analyze). Indeed, when ignoring generics, it is possible to generate the type expression by simply recursively visiting the parents of a given class. Such a simple process cannot be used when generics are involved as the generic parameters can lead to *loops* in the followed path. For example, the complete signature of the Integer class is:

```
class Integer
  extends Number
  implements Comparable<Integer>
```

As a consequence, when producing the type expression for the `Comparable<...>` part of the signature, we cannot make a recursive call to handle the parameter of the `Comparable` interface as it would lead to an infinite loop. Instead, we have to identify the reference to a previously-seen type and refer to it through a type variable. The resulting type will be:

```
[ 'java'io'Serializable
| 'java'lang'Comparable of 'a
| 'java'lang'Integer
| 'java'lang'Number
| 'java'lang'Object ] java_instance as 'a
```

Another problem is the handling of generic bounds. Such bounds allow the Java developer to indicate through the `super` and `extends` keywords lower and upper bound where a generic type parameter is waited. These bounds can be both used at declaration- and use-site, as shown by Listing 13.

```
1  class C<E extends CharSequence> {
2    ...
3  }
4
5  class D {
6    private C<? super String> field;
7    ...
8  }
```

Listing 13. Bounds for generic type parameters.

It should be possible to encode bounds using polymorphic variants combined to variance annotations. Listing 14 and 15 show possible encodings for respectively an `extends` and a `super` bound. However, the question is not only to enhance the current implementation to support generic bounds, but also to be able to emit sensible error messages when constraints are not respected. While generating appropriate messages is not difficult for such simple examples, it becomes difficult when the offending type is nested, which is by definition always the case with generic types.

```
1  type -'v extends
2  let obj : ['Object] extends
3  let num : ['Object|'Number] extends
4  let int : ['Object|'Number|'Integer] extends
5  let str : ['Object|'String] extends
6
7  let f (_ : [> 'Object|'Number] extends) = ...
8  let _ = f obj (* REJECTED *)
9  let _ = f num (* accepted *)
10 let _ = f int (* accepted *)
11 let _ = f str (* REJECTED *)
```

Listing 14. OCaml encoding of an `extends` bound.

```
1  type +'v super
2  let obj : ['Object] super
3  let num : ['Object|'Number] super
4  let int : ['Object|'Number|'Integer] super
5  let str : ['Object|'String] super
6
7  let f (_ : [< 'Object|'Number] super) = ...
8  let _ = f obj (* accepted *)
9  let _ = f num (* accepted *)
10 let _ = f int (* REJECTED *)
11 let _ = f str (* REJECTED *)
```

Listing 15. OCaml encoding of a `super` bound.

10.2 Miscellaneous Enhancements

Another major enhancement, besides generics, to the current implementation would be to allow the developer to extend existing classes and interfaces. As of today, the developer is only able to implement an interface. There is a great practical incentive to make it possible to extend a class: complex event listeners. Indeed, while simple event listeners are specified by interface with one or two methods, complex listeners may contain far more methods. It is usual that the library provides for those listeners *adapter* classes that provides empty implementations for the methods. This allows the developer to extend this class and only override the method(s) she is interested in, without having to take care of the other methods. More generally, the whole purpose of object-oriented design it to provide classes with default implementation to be overridden in order to customize the behavior.

A minor enhancement could also be devised to provide custom support for Java `enum` classes. The current implementation treats `enum` classes as ordinary classes, meaning that the developer has to accesses to the elements of an `enum` through class fields using the `Java.get` function. This is a problem because by doing so, we get a plain Java instance that can then only be tested for equality and cannot be used in pattern matching, leading to code like the following:

```
let open Java in
let state = call "Thread.getState()" th in
if equal state (get "Thread.State.NEW") then
  ...
if equal state (get "Thread.State.BLOCKED") then
  ...
```

while we would like to be able to write the following, also taking advantage of the exhaustiveness and non-redundancy checks of the pattern matching:

```
let state = Java.call "Thread.getState()" th in
match state with
| `NEW -> ...
| `BLOCKED -> ...
...
```

Finally, the typer extension, as the whole OCaml-Java project for that matter, will have to be updated according to the evolution of Java. In particular, the upcoming version of Java will introduce features that should get support in our compiler:

- *lambdas*, that allow to define functions/code blocks without having to go through a inner-classes;
- *default methods*, that allow to define default implementations for interface methods.

Lambdas are particularly interesting, as their inclusion into the Java language will somewhat reduce the semantic gap between the two languages. In order to leverage the full power of lambda-aware libraries, it is desirable to be able to pass an OCaml function where a Java lambda is waited, automatically deriving the necessary wrapper(s) to translate values back and forth.

Default methods shall modify our handling of interfaces, as the developer should not be required to provide a method implementation if a default implementation is available. Still, the developer should be able to override the default implementation when it does not fit her needs. The behavior of the `Java.proxy` function has to be updated to take into account these optional methods.

10.3 Other Languages

We think it could also be interesting to apply the kind of type encoding presented in this paper to other languages. The term "*other languages*" can refer to either *host* languages (OCaml in this article), or *embedded* languages (Java in this article). Regarding embedded languages, we are particularly interested by exploring possible encodings of languages based on structural typing. The idea, in this case, would be to use the set of constructors to encode the set of declared methods rather than the class hierarchy. Such an encoding may use the parameters of the various constructors to encode the parameters types of the methods.

Acknowledgments

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

References

[1] N. Benton, and A. Kennedy: Interlanguage working without tears: blending SML with Java, Proceedings of the fourth ACM SIGPLAN international conference on Functional programming (ICFP '99), 1999.

[2] N. Benton, A. Kennedy, and C. V. Russo: Adventures in interoperability: the SML.NET experience, Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP'04), 2004.

[3] E. Chailloux, G. Henry, and R. Montelatici: Mixing the Objective Caml and C# Programming Models in the .NET Framework, Workshop on Multiparadigm Programming with OO Languages (MPOOL), 2004.

[4] E. Chailloux, G. Henry, and R. Montelatici: Interopérabilité des langages fonctionnels : applications en Objective Caml, Technique et Sciences Informatiques Vol 24/9, 2005.

[5] X. Clerc: the nickel project, `http://nickel.x9c.fr`

[6] X. Clerc: OCaml-Java: OCaml on the JVM. Trends in Functional Programming, Lecture Notes in Computer Science Volume 7829, 2013.

[7] X. Clerc: OCaml-Java: an ML Implementation for the Java Ecosystem. International Conference on Principles and Practices of Programming on the Java platform (PPPJ'13), 2013.

[8] M. Fluet, R. Pucella: Phantom types and subtyping, Journal of Functional Programming, Volume 16, 2006.

[9] R. Hickey: The Clojure programming language. Proceedings of the 2008 symposium on Dynamic language, 2008.

[10] X. Leroy: Camlidl users manual version 1.0, March 1999, `http://caml.inria.fr/camlidl/htmlman`

[11] X. Leroy: The camljava project, `http://forge.ocamlcore.org/projects/camljava/`

[12] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon: The OCaml system release 4.00. Documentation and user's manual, July 2012.

[13] M. Odersky, et al.: The Scala language specification, 2004.

[14] D. Syme, J. Margetson: The F# programming language. `http://research.microsoft.com/projects/fsharp`

[15] J. Yallop: the ctypes project, `https://github.com/ocamllabs/ocaml-ctypes`

[16] SWIG: Simplified Wrapper and Interface Generator, `http://www.swig.org`.

[17] The Apache Foundation: the Thrift project, `http://thrift.apache.org`