Javalib Tutorial

Nicolas Barré

October 30, 2009

Contents

1 Introduction

Javalib is a library written in *OCaml* with the aim to provide a high level representation of *Java* **.class** files. Thus it stands for a good starting point for people who want to develop static analysis for *Java* byte-code programs, benefiting from the strength of *OCaml* language.

This document briefly presents the API provided by *Javalib* and gives some samples of code in *OCaml*.

2 Global architecture

The user interface consists of three modules, *JBasics*, *JCode* and *Javalib*. These modules are the only ones someone should need to write static analysis. The other ones provide lower level functions and should not be used.

2.1 *JBasics* module

This module gives a representation of all the basic types that appear at the byte-code level.

These types are:

- JVM very basic types (such as int, float, double, ...)
- JVM more elaborated types (arrays, objects)
- class names, method signatures and field signatures
- others: constant pool types and stackmaps types

The types representing class names, method signatures and field signatures are abstract. The directives to build them are in *JBasics* and are respectively **make_cn**, **make_ms** and **make_fs**.

This module also provides some sets and maps containers relative to the abstract class names, field signatures and method signatures. These maps and sets are very efficient because they rely on the hidden indexation of each abstract type, thereby reducing the cost of comparison operations between types. Moreover the maps are based on *Patricia Trees* implementation, that is known to be really fast.

2.2 *JCode* module

This module provides a representation of the JVM opcodes and of the code structure at the byte-code level.

It is important to notice that the code is represented by an array of opcodes, and that the position of an opcode in this array corresponds to its relative address in the original binary file (this is the same numbering as **javap**). In *Javalib* representation, an opcode includes an instruction and its arguments.

That's why we needed to introduce a dummy opcode **OpInvalid** to keep the correct numbering.

2.3 *Javalib* module

This is the main module of *Javalib* which contains the definition of classes, fields and methods types, some methods to get a high level representation from binary classes and an internal module *JPrint* to pretty-print or dump every type defined in the three user modules.

3 Tutorial

To begin this tutorial, open an *OCaml* toplevel, for instance using the *Emacs* **tuareg-mode**, and load the following libraries in the given order:

```
#load "str.cma"
#load "unix.cma"
#load "extLib.cma"
#load "zip.cma"
#load "ptrees.cma"
#load "javalib.cma"
```

Don't forget the associated **#directory** directives that allow you to specify the paths where to find these libraries.

You can also build a toplevel including all these libraries using the command **make ocaml** in the sources repository of *Javalib*. This command builds an executable named **ocaml** which is the result of the **ocamlmktop** command.

3.1 Making class names, field signatures and method signatures

Imagine you want to access the method **m:(Ljava.lang.String;)V** and the field **f:I** of the class **A**.

You first need to build the signatures associated to each entity. According to the *Javalib* API you will write:

```
open JBasics
let aname = make_cn "A"
let java_lang_string = make_cn "java.lang.String"
let ms =
make_ms "m" [TObject (TClass java_lang_string)] None
let fs = make_fs "f" (TBasic `Int)
```

3.2 Getting a class representation from a binary file

The methods you need are in the *Javalib* module. You can open this module cause you will need it very often.

```
open javalib
```

Then, you need to build a **class_path** to specify where the classes you want to load have to be found:

```
let class_path = class_path "./" (* for instance *)
```

You can now load the class **./A.class** corresponding to **aname**.

```
let a = get_class class_path aname
```

When you don't need a classpath any more, close it with **close_class_path** if you don't want to get file descriptors exceptions in a near futur.

3.3 Getting fields and methods from a class

You now have the class **a** of type *Javalib.interface_or_class*. You might want to recover its method **m** of type *Javalib.jmethod* and field **f** of type *Javalib.any_field*.

Simply do:

```
let m = get_method a ms
let f = get_field a fs
```

Note: The methods **get_method** and **get_field** raise the exception **Not_found** if the method or field asked for can't be found.

It's important to notice that **a** can be a *Class of jclass* or an *Interface of jinterface* (see type *interface_or_class*), but that the methods **get_method** and **get_field** work equaly on it. That's why **get_field** returns a value of type *any_field* which can be *ClassField of class_field* or *Interface_field of interface_field*. Indeed, according to the JVM specification, we need to make the distinction between interface fields and class fields.

3.4 A more sophisticated example

Now we would like to write a function that takes a **classpath** and a **classname** as parameters and that returns, for each method of this class, a set of the fields accessed for reading (instructions **getstatic** and **getfield**).

Here is the code:

```
open Javalib
open JBasics
open JCode
```

```
let get_accessed_fields (class_path : class_path)
 (cn : class_name) =
 (* We first recover the interface or class associated to the
    class name cn. *)
 let c = get_class class_path cn in
 (* Then, we get all the methods of c. *)
 let methods : jvm_code jmethod MethodMap.t = get_methods c in
 (* For each method of c, we associate a field set containing
    all the accessed fields. *)
   MethodMap.map
    (fun m ->
      match m with
      (* A method can be abstract or concrete. *)
       | AbstractMethod _ ->
         (* An abstract method has no code to parse. *)
          FieldSet.empty
       | ConcreteMethod cm ->
          (match cm.cm_implementation with
          (* A concrete method can be native so that we don't
             know its behaviour. In this case we suppose that
             no fields have been accessed which is not safe. *)
           | Native -> FieldSet.empty
           | Java code ->
             (* The code is stored in a lazy structure, for
                performance purposes. Indeed when loading a
                class the Javalib does not parse its methods. *)
              let jcode = Lazy.force code in
              (* We iter on the array of opcodes, building our
                 field set at the same time. *)
               Array.fold_left
                (fun s op ->
                  match op with
                   | OpGetField (_, fs)
                   | OpGetStatic (_, fs) ->
                  (* We add the field signature in our field set.
                     In this example, we ignore the classes in
                     which the fields are defined. *)
                        FieldSet.add fs s
                   | _ -> s
                ) FieldSet.empty jcode.c_code
          )
    ) methods
```

This method has the signature

```
Javalib.class_path ->
  JBasics.class_name -> JBasics.FieldSet.t JBasics.MethodMap.t
```

3.5 Another use case

Consider the following class written in java:

```
public class TestString{
   public boolean m(String s){
      if (s.equals("str")){
         return true;
      } else{
         return false;
      }
   }
}
```

We see that the method *m* might raise an *NullPointer* exception if we call the method *equals* on an uninitialized string *s*. To avoid this, a good practice is to replace the test **s.equals("str")** by the expression **"str".equals(s)** which will return false rather than raising an exception.

Let's see the bytecode associated to the method *m*, given by **javap**:

```
public boolean m(java.lang.String);
  Code:
   0:   aload_1
   1:   ldc     #2; //String str
   3:   invokevirtual   #3; //Method
           java/lang/String.equals:(Ljava/lang/Object;)Z
```

We will now write a sample of code that detects instructions of type **ldc 'string'** followed by an **invokevirtual** on *java.lang.String.equals* method.

We first need to write a function that returns the next instruction and its program point in a code, given this code and a current program point:

```
let rec next_instruction (code : jopcodes) (pp : int)
  : (jopcode * int) option =
 try
   match code.(pp+1) with
    | OpInvalid -> next_instruction code (pp+1)
    | op -> Some (op,pp+1)
 with _ -> None
```

Now we define a function that takes a *classpath* and a *classname* as parameters and that returns a map associating each concrete method signature to a list of (**int**,**string**) couples representing the program points and the strings on which the *java.lang.String.equals* method is called.

```
let get_equals_calls (class_path : class_path)
  (cn : class_name) =
 (* We first recover the interface or class associated to the
```

```
   class name cn. *)
let java_lang_string = make_cn "java.lang.String" in
let equals_ms =
  make_ms "equals" [TObject (TClass java_lang_object)]
   (Some (TBasic `Bool)) in
let c = get_class class_path cn in
(* Then, we get all the concrete methods of c. *)
let methods : jvm_code concrete_method MethodMap.t =
  get_concrete_methods c in
(* For each concrete method of c, we associate a (int*string) list
   containing all the strings passed as parameters to
   String.equals method, associated to the program point where the
   call occurs. *)
  MethodMap.map
   (fun m ->
    (match m.cm_implementation with
      (* A concrete method can be native so that we don't
         know its behaviour. In this case we suppose that
         no call to String.equals which is not safe. *)
      | Native -> []
      | Java code ->
        (* The code is stored in a lazy structure, for
           performance purposes. Indeed when loading a
           class the Javalib does not parse its methods. *)
        let jcode = Lazy.force code in
        let code = jcode.c_code in
        let l = ref [] in
        (* We iter on the array of opcodes, building
            our list of (int*string) at the same time. *)
          Array.iteri
           (fun pp op ->
             match op with
              | OpConst (`String s) ->
                (* We detect that a string s is pushed on the
                   stack. The next instruction might be an
                   invokevirtual on String.equals. *)
                 (match (next_instruction code pp) with
                   | Some (inst,ppi) ->
                      (match inst with
                        | OpInvoke (`Virtual (TClass cn), ms)
                           when cn = java_lang_string
                             && ms = equals_ms ->
                          (* We add the program point of the
                             invokevirtual and the pushed string
                             in our list. *)
                           l := (ppi, s) :: !l
```

```
                        | _ -> ()
                      )
                    | None -> ()
                  )
                | _ -> ()
              ) code;
            (* We simply return our list, in the reverse order so that
               the program points appear in ascending order. *)
             List.rev !l
     )
    ) methods
```

This method has the signature

```
Javalib.class_path ->
  JBasics.class_name -> (int * string) list JBasics.MethodMap.t
```

We obtain the expected result on the previous class *TestString*:

```
# let cp = class_path ".";;
val cp : Javalib.class_path = <abstr>

# let cn = make_cn "TestString";;
val cn : JBasics.class_name = <abstr>

# let mmap = get_equals_calls cp cn;;
val mmap : (int * string) list JBasics.MethodMap.t = <abstr>

# let l =
    let sk = List.map (ms_name) (MethodMap.key_elements mmap)
    and sv = MethodMap.value_elements mmap in
      List.combine sk sv;;
val l : (string * (int * string) list) list =
  [("m", [(3, "str")]); ("<init>", [])]

# let () = close_class_path cp;;
```