Sawja Tutorial

Nicolas Barré

October 30, 2009

Contents

1 Introduction

Sawja is a library written in *OCaml*, relying on the *Javalib* to provide a high level representation of *Java* byte-code programs. Whereas *Javalib* is dedicated to class per class loading, *Sawja* introduces a notion of program thanks to control flow algorithms. For instance, a program can be loaded using various algorithms like *Class Reachability Analysis* (*CRA*), a variant of *Class Hierarchy Analysis* algorithm (*CHA*) or *Rapid Type Analysis* (*RTA*). For now, *RTA* is the best compromise between loading time and precision of the call graph. A version of *XTA* is coming soon. To get more information about control flow graph algorithms and their complexity, you can consult the paper of Frank Tip and Jens Palsberg[1].

In *Sawja*, classes and interfaces are represented by interconnected nodes belonging to a common hierarchy. For example, given a class node, it's easy to access its super class, its implemented interfaces or its children classes. The next chapters will give more information about the nodes and program data structures.

Moreover, *Sawja* provides some stack-less intermediate representations of code, called *JBir* and *A3Bir*. Such representations open the way to many analyses which can be built upon them more naturally, better than with the byte-code representation (e.g. *Live Variable Analysis*). The transformation algorithm, common to these representations, has been formalized and proved[2].

Sawja also provides functions to map a program using a particular code representation to another.

2 Global architecture

In this section, we present the different modules of *Sawja* and how they interact together. While reading the next sections, we recommend you to have a look at *Sawja* API at the same time.

2.1 *JProgram* module

This module defines:

- the types representing the class hierarchy.
- the program structure.
- some functions to access classes, methods and fields (similar to *Javalib* functions).

[1]F. Tip, J. Palsberg. *Scalable Propagation-Based Call Graph Construction Algorithms.* OOPSLA 2000. See http://www.cs.ucla.edu/~palsberg/paper/oopsla00.pdf.

[2]D. Demange, T. Jensen and D. Pichardie. *A Provably Correct Stackless Intermediate Representation For Java Bytecode.* Research Report 7021, INRIA, 2009. See http://irisa.fr/celtique/ext/bir.

- some functions to browse the class hierarchy.
- a large set of program manipulations.

Classes and interfaces are represented by **class_node** and **interface_node** record types, respectively. These types are parametrized by the code representation type, like in *Javalib*. These types are private and cannot be modified by the user. The only way to create them is to use the functions **make_class_node** and **make_interface_node** with consistent arguments. In practice, you will never need to build them because the class hierarchy is automatically generated when loading a program. You only need a read access to these record fields.

The program structure contains:

- a map of all the classes referenced in the loaded program. These classes are linked together through the node structure.
- a map of parsed methods. This map depends on the algorithm used to load the program (*CRA*, *RTA*, ...).
- a static lookup method. Given the calling class name, the calling method signature, the invoke kind (virtual, static, ...), the invoked class name and method signature, it returns a set of potential couples of (**class_name**, **method_signature**) that may be called.

2.2 *JCRA*, *JRTA* and *JRRTA* modules

Each of these modules implements a function **parse_program** (the signature varies) which returns a program parametrized by the **Javalib.jcode** representation.

In *RTA*, the function **parse_program** takes at least, as parameters, a classpath string and a program entry point. The **default_entrypoints** value represents the methods that are always called by *Sun JVM* before any program is launched.

In *CRA*, the function **parse_program** takes at least, as parameters, a classpath string and a list of classes acting as entry points. The **default_classes** value represents the classes that are always loaded by *Sun JVM*.

JRRTA is a refinement of *RTA*. It first calls *RTA* and then prunes the call graph.

If we compare these algorithms according to their precision on the call-graph, and their cost (time and memory consumption), we get the following order : $CRA < RTA < RRTA < XTA$.

2.3 *JNativeStubs* module

This module allows to define stubs for native methods, containing information about native method calls and native object allocations. Stubs can be stored in files, loaded and merged. The format to describe stubs looks like:

```
Method{type="Native" class="Ljava/lang/String;"
       name="intern" signature="()Ljava/lang/String;"}{
  VMAlloc{
    "Ljava/lang/String;"
    "[C"
  }
}


Method{type="Native" class="Ljava/io/UnixFileSystem;"
       name="getLength" signature="(Ljava/io/File;)J"}{
  Invokes{
    Method{type="Java" class="Ljava/lang/String;"
           name="getBytes" signature="(Ljava/lang/String;)[B"}
    }
}
```

JRTA admits a stub file as optional argument to handle native methods.

2.4 *JControlFlow* module

JControlFlow provides many functions related to class, field an method resolution. Static lookup functions for **invokevirtual**, **invokeinterface**, **invokestatic** and **invokespecial** are also present.

This module also contains an internal module **PP** which allows to navigate through the control flow graph of a program.

2.5 *JBir* and *A3Bir* modules

These modules both declare a type **t** defining an intermediate code representation. Both representations are stack-less. *A3Bir* looks like a three-address code representation whereas expressions in *JBir* can have arbitrary depths.

Each module defines a function **transform** which takes as parameters a concrete method and its associated **JCode.code**, and returns a representation of type **t**. This function coupled with **JProgram.map_program2** can be used to transform a whole program loaded with *RTA* algorithm for example.

2.6 *JPrintHtml* module

This module provides a main function **pp_print_program_to_html_files** to dump a program into a set of **.html** files (one per class) related together by the control flow graph. This function takes as parameters the program, the name of the output directory and a type **info**. The type **info** is used to insert custom annotations at different levels : class, method, field and program point. A value **void_info** is also given and can be used by default.

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"
#load "sawja.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 *Sawja*. This command builds an executable named **ocaml** which is the result of the **ocamlmktop** command.

3.1 Loading and printing a program

In this section, we present how to load a program with *Sawja* and some basic manipulations we can do on it to recover interesting information.

In order to test the efficiency of *Sawja*, we like to work on huge programs. For instance we will use *Soot*, a *Java Optimization Framework* written in *Java*, which can be found at http://www.sable.mcgill.ca/soot. Once you have downloaded *Soot* and its dependencies, make sure that the **$CLASSPATH** environment variable contains the corresponding **.jar** files and the *Java Runtime* **rt.jar**. The following sample of code loads *Soot* program, given its main entry point:

```
open Javalib
let (prta,instantiated_classes) =
  JRTA.parse_program (Sys.getenv "CLASSPATH")
     (make_cn "soot.Main", JProgram.main_signature)
```

It can be interesting to generate the **.html** files corresponding to the parsed program **prta**. We first need to build an **info** type.

```
(* p_class annots a class, saying if it may be instantiated
   or not. *)
let p_class =
  (fun cn ->
    let ioc = get_node prta cn in
      match ioc with
       | Class c ->
          if ClassMap.mem (get_name ioc) instantiated_classes then
            ["Instantiated"] else ["Not instantiatied"]
       | _ -> []
  )

(* p_method annots a method, saying if it is concrete or abstract,
   and if it has been parsed or not (by RTA). *)
let p_method =
  (fun cn ms ->
    let m = get_method (get_node prta cn) ms in
       match m with
        | AbstractMethod _ -> ["Abstract Method"]
        | ConcreteMethod cm ->
          let cms = make_cms cn ms in
          let parse =
            if ClassMethodMap.mem cms prta.parsed_methods then
              "Parsed" else "Not parsed" in
            ["Concrete Method "; parse]
  )

(* There is no field annotation. *)
let p_field = (fun _ _ -> [])

(* There is no program point annotation. *)
let p_pp = (fun _ _ _ -> [])

(* This is the info type. *)
let simple_info =
  { JPrintHtml.p_class = p_class;
    JPrintHtml.p_field = p_field;
    JPrintHtml.p_method = p_method;
    JPrintHtml.p_pp = p_pp }
```

Then we just need to call the printing function:

```
    let output = "./soot"
    let () =
      JPrintHtml.pp_print_program_to_html_files prta
        output simple_info
```

3.2 Transforming a program with *Bir*

In this section we present a sample of code transforming a program loaded with *RTA* to *JBir* representation. The procedure to obtain the *A3Bir* representation is exactly the same.

```
    let pbir = map_program2
      (fun _ -> JBir.transform ~compress:false) prta
```

Warning: Subroutines are not yet handled in *JBir* and *A3Bir*. If some transformed code contains such subroutines, the exception **JBir.Subroutine** or **AB3Bir.Subroutine** will be raised, respectively. However, when transforming a whole program with the above function, no exception will be raised because of the *lazy* evaluation of code.

To see how *JBir* representation looks like, we can pretty-print one class, for instance **java.lang.Object**:

```
let obj = get_node pbir JBasics.java_lang_object
let () = JPrint.print_class (JProgram.to_jclass obj)
    JBir.print stdout
```

Note: We can't dump the whole program using *JPrintHtml* module yet, because it depends on *JCode.code* representation.