Sawja Tutorial

Nicolas Barré Vincent Monfort Pierre Vittet

August 30, 2012

Contents

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 available and provides a way to refine the call graph of a program. 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 to be semantics-preserving[2].

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

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. All modules of *Sawja* are sub-modules of the package module *Sawja_pack* in order to avoid possible namespace conflicts.

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

[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://www.irisa.fr/celtique/ext/bir.

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.

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 class-path 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 class-path 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*.

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.

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.

JBir, JBirSSA, *A3Bir* and *A3BirSSA* 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. *JBirSSA* and *A3BirSSA* are variants which respect the Static Single Assignment (SSA) form.

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.

JPrintHtml module

This module allows, for a given code representation, to dump a program into a set of **.html** files (one per class) related together by the control flow graph.

It provides a functor *Make* that can be instantiated by a module of signature *PrintInterface*. This functor generates a module of signature *HTMLPrinter* containing a function **print__program**.

This module is internally used by the different *Sawja* code representations through a **print__program** function to dump a program using their representation.

The printer for *JCode*, which is a *Javalib* module is defined in *JPrintHtml* using the presented functor. It will be used as example in the tutorial.

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. If you installed sawja with FindLib you should do:

```
#directory "<package_install_path>extlib"
#directory "<package_install_path>camlzip"
#directory "<package_install_path>ptrees"
#directory "<package_install_path>javalib"
#directory "<package_install_path>sawja"
(*<package_install_path> is given by command 'ocamlfind printconf'.
If it is the same path than standard ocaml library just replace by '+'.*)
```

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.

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, the *Java Runtime* **rt.jar** and the *Java Cryptographic Extension* **jce.jar**. The following sample of code loads *Soot* program, given its main entry point:

```
open Javalib_pack
open Javalib
open JBasics
open Sawja_pack
open JProgram
let (prta,instantiated_classes) =
  JRTA.parse_program (Sys.getenv "CLASSPATH")
        (JBasics.make_cms
           (JBasics.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 = "./prta"
let () = JPrintHtml.JCodePrinter.print_program ~info:simple_info prta output
```

Note: The destination directory must exist, otherwise an exception will be raised.

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 = JProgram.map_program2
   (fun _ -> JBir.transform ~bcv:false ~ch_link:false ~formula:false ~formula_cmd:[])
   (Some (fun code pp -> (JBir.pc_ir2bc code).(pp)))
   prta
```

Warning: : Subroutines inlining is handled in *JBir* and *A3Bir* only for not nested subroutines (runtime of version JRE1.6_20 contains a few nested subroutines and next version 1.7 none). 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 = JProgram.get_node pbir JBasics.java_lang_object
let () = JPrint.print_class (JProgram.to_ioc obj)
    JBir.print stdout
```

Or generate the *.html* files corresponding to *JBir* program:

```
let output = "./pbir"
let () = JBir.print_program pbir output
```

Note: If some exceptions occur during *JBir* transformation, the incriminated methods won't have any body in the *.html* representation.

Writing your own HTML printer

Here we study the case of **JCode.jcode** printer, and show in details how it is implemented.

We remember that the module interface to implement is *PrintInterface* whose signature is given below.

```
module type PrintInterface =
sig
  type instr
  type code
  val iter_code : (int -> instr list -> unit) -> code Lazy.t -> unit
  val method_param_names : code program -> class_name ->
     method_signature -> string list option
  val inst_html : code program -> class_name -> method_signature ->
    int -> instr -> JPrintHtml.elem list
end
```

The type **instr** represents the instructions in your code representation type **code**. Here, **instr** corresponds to **JCode.jopcode** and **code** is **JCode.jcode**.

Then, you need to give an iterator on you code structure. This function is really easy to write.

```
let iter_code f lazy_code =
  let code = Lazy.force lazy_code in
    Array.iteri
      (fun pp opcode ->
        match opcode with
          | JCode.OpInvalid -> ()
          | _ -> f pp [opcode]
      ) code.JCode.c_code
```

You also need to provide a function that may associate names to method parameters in the signature. Then, when generating the html instructions you need to be consistent with those names. In our implementation *JCodePrinter* in *jPrintHtml.ml*, we use the source variables names when the local variable table exists in the considered method. If you want to test your printer very quickly, you can define:

```
let method_param_names _ _ _ = None
```

Now, we need to define how to display the instructions in html. In order to do that, some html elements can be created by using predefined functions in *JPrintHtml*. These functions are **simple_elem**, **value_elem**, **field_elem**, **invoke_elem** and **method_args_elem**. The sample of code below will help you to understand how you can use these functions. You are also recommended to read the *API* documentation.

```
open JPrintHtml

let inst_html program cs ms pp op =
  match op with
    | JCode.OpNew ccs ->
      let v = TObject (TClass ccs) in
        [simple_elem "new"; value_elem program cs v]
    | JCode.OpNewArray v ->
        [simple_elem "newarray"; value_elem program cs v]
    | JCode.OpGetField (ccs,fs) ->
      let ftype = fs_type fs in
        [simple_elem "getfield";
         field_elem program cs ccs fs;
         simple_elem " : ";
         value_elem program cs ftype]
    | JCode.OpInvoke ((`Virtual o),cms) ->
      let ccs =
              match o with
              | TClass ccs -> ccs
            | _ -> JBasics.java_lang_object
        and inst =
          Javalib.JPrint.jopcode ~jvm:true op
          in
             [simple_elem inst;
```

```
           invoke_elem program cs ms pp ccs cms;
           method_args_elem program cs ms]
  (*| ... -> ...*)
    | _ ->
      let inst =
        Javalib.JPrint.jopcode ~jvm:true op in
        [simple_elem inst]
```

The html elements have to be concatenated in a list and will be displayed in the given order. The element returned by **simple_elem** is raw text. The element returned by **value_elem** refers to an html class file. The element returned by **field_elem** is a link to the field definition in the corresponding html class file. Field resolution is done by the function **resolve_field** of *JControlFlow*. If more than one field is resolved (it can happen with interface fields), a list of possible links is displayed. The element returned by **invoke_elem** is a list of links refering to html class file methods that have been resolved by the **static_lookup_method** function of **JProgram.program**. The element returned by **method_args_elem** is a list of **value_elem** elements corresponding to the method parameters. They are separated by commas and encapsulated by parentheses, ready to be displayed.

If you don't want any html effect, the above function becomes very simple:

```
let inst_html _program _ _ _ op =
  let inst =
    Javalib.JPrint.jopcode ~jvm:true op in
    [simple_elem inst]
```

Create an analysis for the Sawja Eclipse Plugin

In this section we will use the live variable analysis, included in Sawja as a dataflow analysis example, to create an algorithm that detects unused variable assignment, and turn it into a component of the *Sawja Eclipse Plugin*.

We use the result of the *live variable analysis* associated with the JBir code representation (Live_bir module in Sawja) that returns the live variables before the execution of an instruction. For each instruction **JBir.AffectVar (var,expr)** we check that on the next instruction the variable **var** is alive: if not, it is a dead affectation.

The analysis should notify the programmer in case of a dead variable affectation, as it could be a sign of a bug. As a consequence we want to put warnings on the dead affectation instruction and on the method containing it. We also want to give more verbose information on the result of the analysis, in this case to indicate, for each instruction, which variables are live.

In precedent tutorials we used the *OCaml* toplevel but for this one we want to generate an executable: as a consequence we will use the native ocaml compiler and construct the file **dvad.ml** step by step (bottom up).

The head of the **dvad.ml** file should load the *Javalib* and *Sawja* library packages:

```
open Javalib_pack
open Sawja_pack
```

We first parse the arguments of our executable using the module *ArgPlugin* which is a wrapper to the standard *Arg* module. It will allow us to directly add our executable in the Eclipse plugin, just by dropping it in a folder. In order to automatically analyze all the classes in a project (see documentation of *ArgPlugin*), our code will parse a list of class names.

```
(** [_plugin_exec] parses arguments and executes the analysis for each given
    class file*)
let _plugin_exec =

    (* Arguments values*)
  let targets = ref []
  and classpath = ref ""
  and path_output = ref "" in

    (* Description and instanciation of the arguments*)
  let exec_args =
    [ ("--files", "Class files",
       ArgPlugin.ClassFiles (fun sl -> targets := sl),
       "The class files to pass to the live affectation checker.");
      ("--classpath", "Class path",
       ArgPlugin.ClassPath (fun s -> classpath := s),
       "Locations where class files are looked for.");
    ]
  in
  let usage_msg = "Usage: " ^Sys.argv.(0)^ " [options]" in

    (* Add analysis description and output, and launch parsing of arguments *)
    ArgPlugin.parse
      ("DVAD","Dead Variables Affected Detection: detects variables affected but never read.
      exec_args
      (ArgPlugin.PluginOutput ("--plug_output",(fun s -> path_output := s)))
      usage_msg;
    try
      let cp = Javalib.class_path !classpath in

        (* Launch the analysis on each given class *)
        List.iter
          (fun cn_string ->
             main cp !path_output cn_string (*==> Next function to write *)
          )
          !targets
    with e ->
      (* If an exception is raised, it will appear on the standard
         error output and then in the Error Log view of Eclipse*)
      raise e
```

We then write the main function: its job is to run the analysis on a class and provide the data structure of warnings and information to the Eclipse plugin.

```
(** [main cp output cn_string] loads the class [cn_string], converts it
   in JBir representation, runs the analysis and prints the information
   for the plugin. [cp] is a class_path, [output] a folder path for
   generation of plugin information and [cn_string] the fully
   qualified name of a Java class file.*)
let main cp output cn_string =
  let cn = JBasics.make_cn cn_string in
  let ioc = Javalib.get_class cp cn in
  let bir_ioc = Javalib.map_interface_or_class_context JBir.transform ioc in
  let plugin_infos = run_dead_affect bir_ioc (*==> We need to write this function next*)
  in

    (* Print infos on the current class for the Eclipse plugin*)
    JBir.PluginPrinter.print_class plugin_infos bir_ioc output
```

We create the empty data structure containing the information for the Eclipse plugin and launch the analysis for each method of a class.

```
(**[run_dead_affect ioc] returns the data structure containing
   information for the Eclipse plugin. [ioc] is the interface_or_class
   to check.*)
let run_dead_affect ioc =
  let plugin_infos = JBir.PluginPrinter.empty_infos in
    Javalib.cm_iter
      (fun cm ->
   match cm.Javalib.cm_implementation with
       Javalib.Native -> ()
     | Javalib.Java lazc ->
         let code = Lazy.force lazc in

         (* Launch the live variable analysis *)
         let live = Live_bir.run code in
         let (cn,ms) = JBasics.cms_split cm.Javalib.cm_class_method_signature in
           method_dead_affect
             cn ms code live plugin_infos (*==> We need to write this function next*)
      )
      ioc;
    plugin_infos
```

Finally we write the part of the analysis that checks a method and fills the *plug_info* data structure with warnings and information for the plugin.

```
(**[method_dead_affect cn ms code live plug_info] modifies the data
  structure [plug_info] containing information for the Eclipse
  plugin. [cn] is a class_name, [ms] a method_signature, [code] the
  JBir code of the method and [live] the result of the live analysis
  on the code.*)
let method_dead_affect cn ms code live plugin_infos =

  (*Corner cases: false positive on AffectVar instruction*)
  let not_corner_case i =
    (* Check all AffectVar instructions corresponding to a catch(Exception e) statement*)
    List.for_all
      (fun exc_h -> not(i = exc_h.JBir.e_handler))
      (JBir.exc_tbl code)
  in
  let dead_var_exists = ref false in
    Array.iteri
      (fun i op ->
         (match op with
            | JBir.AffectVar (var,_) when not_corner_case i ->

                (* Is the variable dead at next instruction ?*)
                let live_res = live (i+1) in
                let dead_var = not (Live_bir.Env.mem var live_res) in
                  if dead_var
                  then
                    begin

                      (* We add a warning on the dead affectation instruction*)
                      let warn_pp =
                        (Printf.sprintf
                           "Variable '%s' is affected but never read !"
                           (JBir.var_name_g var), None)
                      in
                        plugin_infos.JPrintPlugin.p_warnings <-
                        JPrintPlugin.add_pp_iow
```

```
                          warn_pp cn ms i plugin_infos.JPrintPlugin.p_warnings;
                        dead_var_exists := true
                    end
            | _ -> ()))
      (JBir.code code);

    (* Fill information for plugin depending on the method analysis result *)
    fill_debug_infos
      !dead_var_exists cn ms code live plugin_infos (*==> Last function to write *)
```

We just have to add a warning and to provide any relevant information from the analysis on a method that contains at least one dead affectation.

```
(** Fill debug information when dead affectations are detected*)
let fill_debug_infos dead_found cn ms code live plugin_infos =
  let _warns =
    if dead_found
    then
      plugin_infos.JPrintPlugin.p_warnings <-
        JPrintPlugin.add_meth_iow
        (JPrintPlugin.MethodSignature "The method contains dead affectation(s).")
        cn ms plugin_infos.JPrintPlugin.p_warnings
  and _infos =
    if dead_found
    then
      begin

      (* We give information on the variables liveness for this
         method*)
      Array.iteri
        (fun i _ ->
           let info_live =
             Printf.sprintf "Live variables: %s\n"
               (Live_bir.to_string (live i))
           in
             plugin_infos.JPrintPlugin.p_infos <-
               JPrintPlugin.add_pp_iow info_live cn ms i plugin_infos.JPrintPlugin.p_infos
        )
        (JBir.code code);
        plugin_infos.JPrintPlugin.p_infos <-
          JPrintPlugin.add_meth_iow
          (JPrintPlugin.MethodSignature "Dead variable affectation(s) found")
          cn ms plugin_infos.JPrintPlugin.p_infos ;
      end
    else
      plugin_infos.JPrintPlugin.p_infos <-
        JPrintPlugin.add_meth_iow
          (JPrintPlugin.MethodSignature "No dead variable affectation found")
          cn ms plugin_infos.JPrintPlugin.p_infos;
  in ()
```

Now we have created the file **dvad.ml**: we can create our executable with the following command:

```
ocamlfind ocamlopt -package sawja -linkpkg -o dvad dvad.ml
```

Finally we can copy our executable **dvad** in the folder of executables as described on the Sawja Eclipse Plugin page.

The implementation of this tutorial is supplied with the *Sawja* library (version > 1.2) as the file **dvad-plugin.ml** in *src/dataflow_analyses*. It also demonstrates how to insert HTML code to display the information on the variable liveness.

Using *formulae* to make assertions

Formulae is a new feature of Sawja 1.4. It provides a way to add some special assertions into a *Bir* representation (JBir, A3Bir. . .) using dedicated Java stub methods. To keep full compatibility with the previous versions of *Sawja* it is disabled by default. You can enable *formulae* by setting the optionnal **formula** argument.

```
open Javalib_pack
open Sawja_pack

(* print the html representation of the class [cn_string] at JBir
  level with the default formula enabled. [cp] is the java classpath and
  [outputDir] is the directory in which the html file will be
  generated. *)

let print cp cn_string outputDir =
  let cn = JBasics.make_cn cn_string in
  let ioc = Javalib.get_class cp cn in
  let bir_ioc = Javalib.map_interface_or_class_context
                  (JBir.transform ~formula:true ) ioc
  in
    JBir.print_class bir_ioc outputDir
```

```
(* Example of use in the same folder as the Java class *)
let main =
  let cp = Javalib.class_path "." in
    print cp "TestFormula" "html_dir"
```

If you set the **formula** argument to **true** without specifying the **formula_cmd** argument, you will be using the default_formulae. It means that the default Java stub methods are going to be used to generate the *formulae*. A Java call to such a method will be replaced in the generated *JBir* code by the *formula*. Those static methods must return **void** and take only a single boolean argument, otherwise, it will not be considered as a *formula*.

The default value offers 3 Java static methods named **'assume'**, **'check'**, **'invariant'** and defined in the Class **'sawja.Assertions'**. The java source file is present in **runtime/sawja/Assertions.java** and will be necessary to compile your Java class using formulae (methods bodies are empty since those methods will not really be called) .

For example the following java code:

```
public class TestFormula{

       public static Object mayBeNull;

       public static Object doit(){
           Object i = mayBeNull;
           sawja.Assertions.check(i!=null);
           return i;
       }

}
```

will be translate without formulae as:

```
 public static java.lang.Object doit ( ) ;

   0: mayinit TestFormula
   1: $bcvar0 := TestFormula.mayBeNull
   2: if ($bcvar0:java.lang.Object == null) goto 5
   3: $T0_13 := 1
   4: goto 6
   5: $T0_13 := 0
   6: mayinit sawja.Assertions
   7: sawja.Assertions.check ($T0_13:I)
   8: return $bcvar0:java.lang.Object
```

but with default formulae it will become:

```
 public static java.lang.Object doit ( ) ;

   0: mayinit TestFormula
   1: $bcvar0 := TestFormula.mayBeNull
   2: nop
   3: nop
   4: nop
   5: nop
   6: nop
   7: FORMULA: sawja.Assertions.check($bcvar0:java.lang.Object != null)
   8: return $bcvar0:java.lang.Object
```

You must pay attention to the fact that the formula directly stores the boolean expression which was given as argument to the Java method. This expression can then be directly manipulated in different analyzes to make assumptions. The generated ‘nop’ instructions are replacing the instructions which have been used to create the formula.

In the current implementation, the expression obtained when working with the **A3bir** representation is quite limited (it is often a simple temporary variable, containing the result of the expression). It will be completed in a next release.

You can also create your own *formula handler*, using the class and static methods you want (methods must return void and take only a single boolean argument). You just have to do the as follows:

```
open Javalib_pack
open Sawja_pack

(* print the html representation of the class [cn_string] at JBir
  level with a personnalized formula enabled. [cp] is the java
  classpath and [outputDir] is the directory in which the html file
  will be generated. *)

let print cp cn_string outputDir =
  let cn = JBasics.make_cn cn_string in
  let ioc = Javalib.get_class cp cn in
  let fhandler =
   let cn_formula_cl = JBasics.make_cn "MyFormulaClass" in
   let ms1_formula = JBasics.make_ms "myFormulaFun1" [JBasics.TBasic `Bool] None in
   let ms2_formula = JBasics.make_ms "myFormulaFun2" [JBasics.TBasic `Bool] None in
       [JBasics.make_cms cn_formula_cl ms1_formula;
        JBasics.make_cms cn_formula_cl ms2_formula ]
  in
  let bir_ioc = Javalib.map_interface_or_class_context
                  (JBir.transform ~formula:true ~formula_cmd:fhandler) ioc
  in
    JBir.print_class bir_ioc outputDir
```

You can then use the printing functions on Java code containing a call to **"MyFormulaClass.myFormulaFun1"**, you will see that the call is replaced by a *formula* instruction.