A static liveness analysis for the Ruby language

Nicolò Marchi - VR365684
Alessandro Gottoli - VR352595

March 20, 2014

Contents

1 Introduction

The objective of this project is to statically analyse the liveness of variables in Ruby. Ruby is a very complex language because of its very rich syntax and its semantics based on uncommon special cases that allow the user to write the same program in several different ways. For these reasons Ruby is highly different from common and largely used programming languages. In order to avoid the language complexity, and work on a "linearized" version of the Ruby language (that is, a transformation of all the redundant constructs in a precise defined version) we decide to use the Ruby Intermediate Language (RIL), that contains a parser and an intermediate representation of the Ruby language, which is designed to be extended easily and to analyse and transform Ruby source code. RIL is written in OCaml [4], and it's presented in the article [3]. In addition, inside the paper a `nil`-pointer analysis is presented, corrected and improved by the work [1] of the group of colleagues composed by Federico De Meo, Oscar Maraia and Fabio Signorini. We based our liveness analysis on their `nil`-pointer analysis implementation.

1.1 Liveness Analysis

In data-flow analysis, *Liveness Analysis* is an inquiry that answer to "is the value of this variable needed?" question. To do this, we compute for every instruction the state of each variable of the program that can be either live or dead. The criteria for assigning a fact (*Live* or *Dead*) to a variable is given by:

- *live*: when there's a reference to the variable inside the instruction;
- *dead*: when the variable is assigned.

There are two types of liveness, semantic and syntactic.

A variable x is semantically live at node n if there are some execution sequence starting at n whose (externally observable) behavior can be affected by changing the value of x. This kind of liveness is undecidable in general, and it's concerned with the execution behavior of the program.

In syntactical liveness analysis a variable is live at a certain node if there is a path to the exit of the flow-graph along which its value may be used before it is redefined. Syntactic liveness is concerned with properties of the syntactic structure of the program, and it's an approximation of semantic liveness. This generate imprecision of the analysis that means we safely overestimate liveness and we could have variables set to live instead of dead, but never we set dead a variable that is live. That means:

$$\text{semantics-live}(n) \subseteq \text{syntactics-live}(n)$$

The syntactic version is decidable, and it's the kind of analysis that we perform. Indeed liveness is a backwards data-flow analysis, in which we propagate the future information backwards throw the program to discover which variables are live. This propagation is defined by the set called *in-* and *out-* live which are respectively the set of the variables that are live immediately before and immediately after a node. Since each node can have more predecessors and successors in the data-flow graph, we perform the union of the sets in the confluences.

There are a lot of algorithms for computing liveness, like iterative approaches. In our work we used an approach based on a fix-point algorithm, that we will examine in depth in Section 3.2.

2 Ruby Intermediate Language (RIL)

Ruby Intermediate Language is an open source framework [3] embedded in another project, called *DRuby*[2]. As already explained, this framework afford a data-flow engine and a transformation of the Ruby code in an intermediate version.

In fact *Ruby* is an object oriented, dynamic scripting language inspired by *Perl*, *Python*, *Smalltalk* and *LISP*, and aims to "feel natural to programmers" by providing a rich and ambiguous syntax, and a semantics that includes a significant amount of special cases and implicit behaviors. For that complexity, writing a static analysis tool for Ruby could be very difficult, and in such a situation RIL can help us to easily handle Ruby code. The major advantages provided by RIL are:

- the parser of RIL which is separated from the Ruby interpreter, and it uses a Generalised LR (GLR) grammar, that is eventually easy to extend;
- as we previously mentioned, RIL translates most of the redundant syntactic forms into one common representation, as shown it in the Listing 1, in the `if` statements.
- RIL takes the Ruby's evaluation order and explicits it by assigning intermediate results to temporary variables called `__tmp_i` (where `i` changes depending on the number of already existing temporary variables), making flow-sensitive analyses simpler to write;
- RIL shows much of Ruby's implicit semantics, for example it replaces the empty method bodies by a `return nil` statement to indicate their behavior.

A practical example of these transformations is given by the following listings. First, we can see some Ruby statements.

```ruby
###### Classic If ######

x= 1
if x > 2
   x+1
else
   x-1
end

###### Inline If ######

var =  1
print "else branch\n" unless var
```

Listing 1: Original Ruby Code

Now we can see the transformation made by RIL:

```
(* Transformation of the classic If *)

x = 1
__tmp_1 = x.>(2)
if __tmp_1 then
  __tmp_2 = x.+(1)
else
  __tmp_3 = x.-(1)
end

(* Transformation of the in-line If *)

var = 1
if var then
  nil
else
  print(%{else branch\n})
end
```

Listing 2: Transformed RIL code

To further clarify see [1] and [3].

3 Liveness Analysis using RIL

The RIL framework provides instruments to developers for statically analysing Ruby code. For executing a data-flow analysis, it's possible to use the RIL's built-in data-flow analysis engine. Note that developers must use a precise signature in the analysis specifications in order to take advantage of this engine. The signature is the following:

```
module type DataFlowProblem =
sig
    type t                          (* abstract type of facts *)
    val empty : t                   (* initial fact for stmts *)
    val eq : t -> t -> bool         (* equality on facts *)
    val to_string : t -> string     (* to string function for facts *)

    val transfer : t -> stmt -> t   (* transfer function *)
    val join : t list -> t          (* join operation *)
    val meet : t -> t -> t          (* meet operation *)
end
```

After the definition of this "contract" and the implementation of the required functions, this `DataFlowProblem` module could be used by RIL to perform a data-flow analysis. RIL includes basic support for forwards and backwards analysis, by the determination of a *fix-point*, reached by comparing old and new data-flow facts using the signature `eq` method. Other main functions used by the fix-point algorithm are:

- `join`: this function literally joins maps of different facts that originate from various branches of a statement; for example an `if` statement consists of two branches, a `then` branch and an `else` branch. Those branches are statements too and after their analysis we have to merge the resulting facts. For this operation the `join` function comes out, and with the `meet` function it decides how to put together different facts on the same variable;

- `meet`: this function joins different facts; it takes as input two facts, and gives out the "merge" of the two;
- `transfer`: is a function that given as input the data-flow facts map containing the facts already computed and related to the execution flow preceding a particular statement of the program, returns the new facts relative to the execution flow resulting as output from the statement in analysis; we deeply describe this function in Section 4.

In our liveness analysis we trace both live variables and dead variables that occur in each statement. We decide to do that for a better comprehension of the execution flow of the Ruby code and in the testing phase for inspecting our liveness analysis.

After these definitions, we invoke the RIL's data-flow engine, to start the computation of the data-flow analysis. In doing this we have first to create the *Control Flow Graph* of the input Ruby source code with these instructions:

```
let main fname =
 let loader = File_loader.create File_loader.EmptyCfg [] in
 let s = File_loader.load_file loader fname in
 let () = compute_cfg s in
 let () = compute_cfg_locals s in
 ...
```

The input for the analysis is the name of a file, which is then parsed, transformed, and printed back to `stdout`. First, we use RIL's `File_loader` module to parse the given file, which is subsequently translated into the RIL intermediate code binding the result to `s` which at this point contains the entire transformed code of the program, representing the root of the abstract syntax tree (AST). Finally the entire AST is populated by adding a node for each statement of the program.

After that we have to call the fix-point algorithm for computing the facts on the Control Flow Graph. We have to instantiate the data-flow analysis engine with our data-flow problem, previously implemented looking at the contract already shown. We call the execution of the fix-point algorithm on `s`.

```
module Liveness = Dataflow.Backwards(LivenessAnalysis)

    let ofs, ifs = Liveness.fixpoint s in
...
```

Since our analysis is backwards, and the `nil`-analysis implemented and modified by the other group is forwards, we initially decided to trust the already implemented backwards fix-point algorithm and use it for our data-flows analysis. In the test phase however we found out that this algorithm behaved badly, as discovered by the other group for the forwards fix-point algorithm. So, we decided to re-implement the fix-point algorithm as we will describe in Subsection 3.3.

After the execution of the fix-point, the program receives as output two hash tables, one called `ifs`, that represents the live-in table of variables and the other called `ofs`, the live-out table of variables. These tables are composed by a statement `stmt` as key, and a `StmtMap` as value, which has the string of a variable as key and the fact related to that variable as value.

With this two tables, the procedure prints out the contents of the tables, and formats all for matching each statement with the facts about the variables in it.

```

Live In Variables Table:

(1) | a = 1                                  | Math,              |
(2) | b = 2                                  | Math, a,           |
(3) | c = 3                                  | Math, a, b,        |
(4) | d = 4                                  | Math, a, b, c,     |
(5) | for j i in [[a, b], [c, d], [c, d]] do | Math, a, b, c, d,  |
(7) |    Math.max(i, j)                      | Math, a, b, i, j,  |
(8) |    c = 1                               | Math, a, b,        |
(9) |    d = 2                               | Math, a, b, c,     |
(5) | end                                    |                    |
--------------------------------------------

Live Out Variables Table:

(1) | a = 1                                  | Math, a,           |
(2) | b = 2                                  | Math, a, b,        |
(3) | c = 3                                  | Math, a, b, c,     |
(4) | d = 4                                  | Math, a, b, c, d,  |
(5) | for j i in [[a, b], [c, d], [c, d]] do |                    |
(7) |    Math.max(i, j)                      | Math, a, b,        |
(8) |    c = 1                               | Math, a, b, c,     |
(9) |    d = 2                               | Math, a, b, c, d,  |
(5) | end                                    |                    |

---------------------------------------------
Liveness analysis complete.
```

Listing 3: Output of the Liveness Analysis

In Listing 3, we can see another feature of Ruby. The `Math` class, that is a default class already implemented in the API of Ruby is treated like a common variable. That because for Ruby this is an already existing variable with inside an instance of the class `Math`, and always thankfully to the power of Ruby, if we re-assign the variable `Math` we lose the reference of that class and we cannot use its methods inside the same execution of Ruby anymore.

3.1 Our `LivenessAnalysis` module

Here it's possible to inspect our implementation of the `DataFlowProblem` for the Liveness analysis. First we can see the definition of the type `t` of data-flow facts as a map from variables names (strings) to *facts*, which are either `Live` or `Dead`:

```ocaml
module LivenessAnalysis = struct

 type fact = Live | Dead
 type t = fact StrMap.t

 let top = StrMap.empty

 let eq t1 t2 = StrMap.compare Pervasives.compare t1 t2 = 0

 let fact_to_s = function Live -> "Live" | Dead -> "Dead"
 let to_string t = strmap_to_string fact_to_s t
```

Next, we define the other requested functions that are:

- `top`: represents the empty map;
- `eq`: compares two maps in each entry of the map, comparing keys and values;
- `fact_to_s`: matches a fact with its string representation;
- `to_string`: prints out the map of facts.

Listing 4 shows the definition of functions `join` and `meet` previously mentioned, plus other support functions used during the execution of this precise `DataFlowProblem`. There is a distinction between the `meet` function for the `if` statement, and the same function for the other statements, and the relatives `update` functions for the map.

```
let meet_fact t1 t2 = match t1,t2 with
  | _, Live ->  Live
  | _, Dead -> Dead

let meet_fact_IF t1 t2 = match t1,t2 with
  | _, Live -> Live
  | Live, _ -> Live
  | Dead, Dead -> Dead

let update s v map =
  let fact =
    try meet_fact (StrMap.find s map) v
    with Not_found -> v
  in
  StrMap.add s fact map

let update_IF s v map =
  let fact =
    try meet_fact_IF (StrMap.find s map) v
    with Not_found -> v
  in
  StrMap.add s fact map
```

Listing 4: Different `meet` functions

The `meet` semi-lattice function considers the new facts computed on a statement more important than the previous ones. A particular case is the `if` statement, where the `meet` function is different (called `meet_fact_IF`), with a priority based on `Live` $\prec$ `Dead` as presented Listing 5:

```
j = 0                         # j dead
if guard then                 # j dead or live? *
    j = 1                     # j dead
else
    puts j                    # j live
end
puts j                        # j live

# * -> live because the assignment at line 2
# has to be read by the puts at line 15 if guard = false
```

Listing 5: Particular case for the `meet` function

Finally, the `join` function joins the facts originate from two branches of a decision statement, or the facts generated by the iteration of a loop statement.

As shown, the function receives in input a list of two maps and it joins the two maps using the `update` function expressly created for joining two separated branches of different iterations. When the function meets two facts, as said it uses a priority based on `Live` $\prec$ `Dead`. In fact the syntactic liveness analysis overestimates the live variables, but never underestimates the dead variables.

```
let join lst =
    let map1 = (fun acc map ->
                  StrMap.fold update_IF map acc
              ) (List.nth lst 0) (List.nth lst 1) in
        let map2 = (fun acc map ->
                      StrMap.fold update_IF map acc
                  ) (List.nth lst 1) (List.nth lst 0) in
            (fun acc map ->
                  StrMap.fold update_IF map acc) map1 map2
```

The `transfer` function wil be explained in Section 4.

3.2 RIL's Backwards fix-point algorithm

The RIL's implementation of the backwards fix-point algorithm was not correct. We discovered bad behaviors during the execution, most of them due to loops in the Ruby code, like `while` and `for` statements and decisional statements like `if` and `case`. For example, studying the `if` construct, we discovered that the fix-point algorithm could not maintain the relationship between the *true* and the *false* branches, and it simply treats them as two sequential block of code losing the natural if semantics. The problem here is that the algorithm uses a queue data structure which loses the connection between the statements. In Appendix A we show the RIL's fix-point code.

First of all, the fix-point function receives as input `s`, that is the main statement. It starts with the creation of a queue, and with an iteration over the statements set, push every statement in the queue, and creating a new entry in the hash table called `in_tbl` using the statement itself as key and the empty map of facts as value.

The next code line is a while loop that checks if the queue is empty. If the guard is satisfied the algorithm enters inside the body of the loop. It pops a new statement from the queue, and it retrieves from the `out_tbl` hash table the map of facts relative to the statement in exam, and puts this map in a list. After that the maps inside the list are joined. The result of the join operation is put in the `in_tbl` hash table, for creating the in-table for the next iteration of the algorithm.

Starting from the last computed map of facts, the algorithm call the `transfer` function, for computing the new facts relative to the statement that is currently under examination. The fix-point at this time retrieves the old facts from the `out_tbl` and compares the map of the newly computed facts with the old one. In case of equality nothing happens, otherwise the fix-point iterates the set of precedent statements or this was supposed, because during our test we discovered that the fix-point algorithm completely loose the iteration. That cause a big lack of information, and a wrong computation of the facts.

However, at the end of the computation the function returns two hash tables, that are:

- `in_tbl`: is the hash table containing the facts before the execution of the fix-point, and it has the statement as key and the map of facts as value;
- `out_tbl`: is the hash table containing the facts after the execution of the fix-point, namely the output facts computed at the statement; the structure of the entries are equal to the `in_tbl`.

We decided to rewrite completely the code of the fix-point algorithm, in a more efficient way, recursive and exact, as described in Subsection 3.3.

3.3 Our implementation of fix-point

In Appendix B we show our code for the fix-point algorithm. The idea of the function is quite simple. Our fix-point function takes as input the same parameters as the previous one. In its body are created the input and output tables (which are the same as the other fix-point function), and we initialise the `in_tbl` with an empty map.

After that we call a recursive auxiliary function, `super_fixpoint`, that computes the facts more efficiently. `super_fixpoint` updates the hash tables with the facts related to the children of the input statement. In particular the new facts are calculated from the recursive calls carried out by the `super_fixpoint` procedure on the children themselves.

This function makes a pattern matching on the input statement because, depending on the statement, we have to do different things.

3.3.1 Sequence of statements

As we discovered inside the RIL code, the "top" statement considered is the Seq statement, which is a list of other statements. This is the first statement that we consider. However, to achieve a backwards analysis we have to reverse the list of statements, because we have to start from the bottom and coming to the top with the analysis. To do this we call the `List.rev` function and, the `super_fixpoint` function on every statement in the reversed list.

```
| Seq(list) ->
  (* the starting input facts are the input facts of the seq *)
  let newfacts = ref !ifacts in
  (* we reverse the list for having a backward analysis *)
  let rev_list = List.rev list in
  (* iterate over it to calculate the facts of all the stmt of the list *)
    List.iter (fun x ->
        (* replace the old in-facts with precedent stmt out-facts *)
        Hashtbl.replace in_tbl x !ifacts;
          (* calculate this stmt newfacts ( = out-facts table) and set in out-tbl *)
        newfacts := super_fixpoint x in_tbl out_tbl;
        Hashtbl.replace out_tbl x !newfacts;
      (* these facts become the in-facts for the next stmt *)
        ifacts := !newfacts
    ) rev_list;
  (* out-fact table of the stmt is exactly the out-facts table of the last stmt analysed *)
  !newfacts
```

3.3.2 Decisional statements

The second case considered is the `If` statement. The analysis of this statement is achieved by recursively calling the auxiliary fix-point function on every branch of the statement. Since both the branch-statements are initially recognised as `Seq` by the auxiliary function, we don't have to reverse the elements of the statements. The results of the auxiliary function calls on the branches are put into a list, that is joined for having an unique result map of facts of the branches. Using the result map we call the `transfer` function, that will take care of computing the guard facts.

```
| If(_, t, f) ->
  (* the guard is ignored because analysed in the transfer *)

  (* set the in-facts of both branches *)
  Hashtbl.replace in_tbl t !ifacts;
  Hashtbl.replace in_tbl f !ifacts;
  (* calculate the out-facts of both branches and save them *)
  let t_facts = super_fixpoint t in_tbl out_tbl in
  let f_facts = super_fixpoint f in_tbl out_tbl in
  Hashtbl.replace out_tbl t t_facts;
  Hashtbl.replace out_tbl f f_facts;

  (* join the 2 out-table following the if semantics, plus calculate the guard *)
  DFP.transfer (DFP.join (t_facts :: (f_facts ::[]))) stmt
```

The `Case` statement is interesting. This statement is composed by a guard and a list of guard-statement pairs that represent the different execution flows corresponding to the values assumable by the upper guard. Therefore, we have to consider all the guards and all the statements of the construct. In order to do that, we write into a list the default case (if present) followed by all the statements inside the `Case` in a reverse order. Now the algorithm starts calling the auxiliary fix-point function for computing the facts for every `when` case backwards, in a reverse order, using a high-order function (called `fold_left`) typical of the functional languages, that allows us to apply a function on every element of the list and accumulate the results in an *accumulation variable*. The reason for doing that is that the modification and use of the same variable in the various statement of the case allows us to consider every fact in those statements. Finally we apply the join operator between all the maps in `finalfacts` because we can not know statically which branch will be taken at execution time and we must assume that each of them is a possible candidate.

```
| Case (b) ->
  (* st is the list of all branch in reverse order *)
  let default = b.case_else in
  let st = match default with
    | None -> []
    | Some s -> s::[]
  in
  let st = List.fold_right ( fun (_, s) acc -> s::acc ) b.case_whens st in

  (* finalfacts will contain for each when's stmt what we know after having analysed it *)
  let finalfacts =
  (* x is each stmt in each branch of the case *)
  List.fold_left ( fun acc x ->
    (* set the in-facts of each stmt with what we know before the case stmt *)
    Hashtbl.replace in_tbl x !ifacts;

    (* calculate the out-facts table of the when stmt and save it in out_table *)
    let newfacts = super_fixpoint x in_tbl out_tbl in
    Hashtbl.replace out_tbl x newfacts;
    (* accumulate all the out-facts *)
    newfacts :: acc
  ) [] st
  in
  (* join all branch out-facts *)
  let tmp_fact = List.fold_left ( fun acc x ->
        DFP.join (acc :: (x :: []))
  ) (List.hd finalfacts) (List.tl finalfacts)
  in
  (* calculate case out-facts *)
  DFP.transfer tmp_fact stmt;
```

3.3.3 Loop statements

While and For are the loops that we take into account. Since these are potentially infinite loops, it is not possible to analyse every execution flow, however we can still reach a fix-point. This is achieved by repeating the computation of the facts of loop body (which could be any kind of statement) until the observed facts for two consecutive iterations are equals. This case was one of the weaknesses of the previous algorithm. To compute the facts through the iterations of cycles we call a specifically created do_while function, which takes two functions as input: one it's the function for the guard of the do_while, and the other one it's the body of the loop. The do_while function saves the existing facts and calls the super_fixpoint function for computing the new ones. If the previous and the new facts are equals, we reached a fix-point, and we can now use the map of facts to execute the transfer function on the guard/guards.

```
| For (_, _, b)
| While(_, b) ->
  (* we analyse only the body because the guard is analysed in the transfer *)

  (* b_facts contains what we know before analysing the for/while *)
  let b_facts = ref !ifacts in
  let old_facts = ref DFP.empty in
  (* iteration while we reach fix-point *)
  do_while (fun () ->
    (* replace in-fact with out-facts of precedent iteration *)
    Hashtbl.replace in_tbl b !b_facts;
    old_facts := !b_facts;
    (* calculate the body out-facts *)
    b_facts := super_fixpoint b in_tbl out_tbl;
    (* calculate loop out-facts *)
    b_facts := DFP.join (!ifacts :: (!b_facts ::[]));
    b_facts := DFP.transfer !b_facts stmt )
    (* check if we reached fix-point *)
    (fun () -> not (DFP.eq !old_facts !b_facts)
  );

  (* save out-facts of the body *)
    Hashtbl.replace out_tbl b !b_facts;

    DFP.join (!ifacts :: (!b_facts ::[])) (* returns a StrMap *)
```

3.3.4 Method Definition

We decide to consider the method definition, and statically analyse the liveness inside the method definition code. To achieve this result, we have to take care of the method definition separately from the rest of the code because the scope of the instructions inside the method code is different from the scope of the rest of the program, so if there are variables with the same name, these are not to be computed in the same table of facts. Therefore our analyser extracts from the Ruby programs all the method definitions and puts all the method definition statements inside a list before calling the fix-point algorithm on all the Ruby programs.

We then iterate this list and execute the fix-point function on all the statements inside the list independently, and print to the user the in-table and out-table for all the method definitions. Eventually we call the fix-point algorithm on all the input programs, and when we find a statement that is a method definition, we basically skip this statement. and skip all the method definition statements we find.

```
(* save all the stmt reached by a stmt in a list and return it *)
let rec acc_stmt todo visited =
  StmtSet.fold (fun stmt acc ->
      match StmtSet.exists (fun x ->
        (* checks if the stmt exists in the list *)
        (string_of_cfg x) = (string_of_cfg stmt)
      ) acc with
      (* if present, do nothing and return the list unaltered *)
      | true -> visited
      (* if not present, add it to the list and analyse the successors *)
      | false -> acc_stmt stmt.succs (StmtSet.add stmt acc)
  ) todo visited


(* find the method def stmt and save them in a list in a reverse order *)
let find_def stmt =
  StmtSet.fold (fun x acc ->
      match x.snode with
      | Method(_,_, s) -> print_stmt stdout s; s::acc
      | _ -> acc
  ) (acc_stmt (StmtSet.add stmt StmtSet.empty) StmtSet.empty) []


(* analyse methods def *)
let list_def = find_def s in
    (* for every def found we do the analysis and print the result *)
    List.iter (fun x -> let o, i = Liveness.fixpoint x in
        print_string "Live In Variables Table of Method Definition: \n \n";
        justif (print_var_table x i 0);
        print_string "--------------------------------------------\n\n";
        print_string "Live Out Variables Table of Method Definition: \n \n";
        justif (print_var_table x o 0);
        print_string "--------------------------------------------\n\n";
    ) list_def;
```

3.4 Our update methods

For having a better readability of the `transfer` function OCaml code, we decided to manage all the types of the grammar provided by RIL through some `update_*` function, where the $*$ corresponds to the type of expression we need to visit.

Those functions receive as input the map of facts that the `transfer` function is computing, the fact (that could be *Live* or *Dead* dependently if we are updating a used variable or an assigned variable) and the expression to match and update inside the map.

In respect to the RIL's grammar we have an update function for:

- *expr*: it represents a general expression and is composed by a literal or an identifier;
- *identifier*: it represents an identifier and is composed by local variables, constant variables, or tuples of other expressions;
- *literal*: it represents a literal (e.g. numbers), and is composed by literals, pairs of expressions, arrays of expressions or ranges of expressions (like `(0..5)`, that represent an interval from 0 to 5);
- *formal_param*: it represents the parameters of the `for` cycle and is composed by formal block ids, formal stars and formal tuples of variables;
- *star_expr*: it represents a star of expressions and is composed by other expressions or other star_expr;
- *tuple_expr*: it represents a tuple of expressions, and is composed by expressions, star of expressions, tuple of expressions or star of tuples of expressions;
- *lhs*: it represents the left-hand side of an assignment and it can be an identifier, a tuple of lhs or a star of identifiers.

Furthermore there are others functions for the `option` case; these functions take care of the cases we're the expression (or other type of it) it's present or not, and if it's present call the adapt update function.

4 Statement analysis by our `transfer` function

In this section we describe in more depth the execution of the `transfer` function on the different statements of Ruby, to capture all the modifies to the variables inside the statements.

4.1 Assign

The *Assign* statement (`Assign of lhs * tuple_expr`) is composed of `lhs` (left-hand side) and a `tuple_expr` (the assigned value). The first operation is the elimination of all the variables in `lhs` and this is done by the `update_lhs` method which sets to dead the variables in the map. There are different cases for the assignment given by the type of the right-hand side. This determines either the different methods to call or no method call in case of `#literal`, `` `ID_True ``, `` `ID_False `` and `` `ID_Nil ``.

```
(* if stmt is an assign with right hand side not contains variables *)
| Assign(lhs , #literal)
| Assign(lhs , `ID_True)
| Assign(lhs , `ID_False)
| Assign(lhs , `ID_Nil) ->
    (* we analyse the left hand side and set it to dead *)
    update_lhs lhs Dead map

(* if it is an assignment with a variable as rhs *)
| Assign(lhs, (`ID_Var(_, _) as var)) ->
    (* we set the lhs to dead, and set the rhs variable to dead *)
    let map = update_lhs lhs Dead map in
    update_expr var Live map

(* if it is an assignment with a composite rhs *)
| Assign(lhs, (`Tuple(_) as tup)) ->
    (* we set the lhs to dead, and analyse the rhs to set to live *)
    let map = update_lhs lhs Dead map in
    update_tuple_expr tup Live map
```

4.2 Expression, If and While

Given our implementation of the fix-point, we consider these three statements in the same way. The *Expression* statement (`Expression of expr`) is composed of an expression. The *While* statement (`While of expr * stmt`) is composed of an expression (guard) and a statement (body of loop). The *If* statement (`If of expr * stmt * stmt`) is composed of an expression (guard) and two statement (then and else branch). Since the statements are analysed by fix-point algorithm, we consider only the guard. How we can see from the code below, we call the `update_expr` method with *Live* as fact because the guard is read. If the guard is a compare of values, or other method calls, it has a special treatment from RIL. The execution of the method is extract and the result putted inside a temporary variable. This variable will be the guard of the `If` statement. This method checks the *expr* type and if it's a variable, add it to out-live set variable.

```
(* if it is a case in which there is only an expr to analyse *)
| Expression(e)
| If(e,_,_)
| While(e, _) ->
    (* we set the expr to live because it is read *)
    update_expr e Live map
```

4.3 For

The *For* statement (`For of block_formal_param list * expr * stmt`) is composed of a list of formal parameters, an expression and a statement consisting the loop body. This isn't the classical `for` of Java or C, this is analogue at the `for each` statement defined in that languages. We consider only the first two arguments and we call the `update_formal_param` to set to dead all the variables that are assigned by the loop. After that we call `update_expr` to set to live the value read in the guard of the loop. How happens for the `while` and `if` guard, this is valid also when the guard is composite, because RIL create a temporary variable and assign to it the composite guard and therefore treat it how an assignment and read only the variable as simple guard. Note that also in this statement, the body of the loop is ignored because it is treated by the fix-point algorithm.

```
| For(p, e, _) ->
    (* we set to dead all the formal parameters *)
    let map = List.fold_left(fun acc x ->
        update_formal_param x Dead acc) map p in
    (* and to live the expression of the for *)
    update_expr e Live map
```

4.4 Case

The *Case* statement (`Case of case_block`) consists in an expression (guard) and a expression-statement pair list ((tuple_expr * stmt) list) which are the different branches corresponding to the value assumed by the guard at execution time, this is recognisable in the Ruby code by the word `when`. Since all the variables are read, we add all the variables to the live-out set of the statement. To do this we call `update_tuple_expr` on all the guards and after call `update_expr` on the general guard. How in other statement analysed yet, we treat only the guards, because the statements are treated by fix-point algorithm.

```
| Case(all) ->
    (* we set to live all the guards (whens and case) *)
    let map = List.fold_left (fun acc (s, _) ->
        update_tuple_expr s Live acc) map all.case_whens in
    update_expr all.case_guard Live map
```

4.5 MethodCall

The `MethodCall` statement (`MethodCall of lhs option * method_call`) is composed of a `lhs option` (where the result is saved) and the structure `method_call` which contains method target, method name, arguments and code block. In this statement we start the analysis from the `lhs option` and since it could be an assignment we call the `update_lhs_option` method to set them to dead. After that we analyse all the arguments and target and set they to live with `update_star_expr`. Since Liveness analysis is a intra-procedural analysis we don't analyse the code inside the called method.

```
| MethodCall(lhs_o, {mc_target = target; mc_args = args} ) ->
    (* we set the lhs to dead *)
    let map = update_lhs_option lhs_o Dead map in
    (* we analyse the arguments of the function and set to live the variables *)
    let map = List.fold_left (fun acc x -> update_star_expr x Live acc) map args
     in
    (* if present, the target of the function is set to live *)
    (* note: classes in Ruby are variables, so we don't care what is the target *)
    update_expr_option target Live map
```

4.6 Yield

The *Yield* statement (`Yield of lhs option * star_expr list`) is a special Ruby statement. In Ruby we can define a block enclosed within braces () and assign a name to it. A block is always invoked from a function with the same name of the block. For invoke a block, we use the `yield` statement. This statement is composed of `star_expr list` that is the value returned by the `yield` to block and a `lhs option` that saves the values. For this reasons we set to live all the variables returned by `yield` and to dead the variables that save the values. How occurs for the statement before, in case of composite expressions, RIL transforms the code creating new temporary variables to assign this expression which are analysed as assignment statement.

```
| Yield(lhs_o ,args) ->
    (* we set to live the values that the yield pass to the block *)
    let map = List.fold_left (fun acc x -> update_star_expr x Live acc) map args
     in
    (* and to dead the variable that save the return of the yield *)
    update_lhs_option lhs_o Dead map
```

4.7 Return

The *Return* statement (`Return of tuple_expr option`) simply returns values from the method where it is, for this reason we have only read operations and we set to live all the variables calling the `update_tuple_expr` method. How we seen before, in case the `yield` return a composite expression RIL create a new temporary variable and assign to it the expression (that are analysed as assignment) and return the new variable.

```
| Return(s)->
    (* we set to live the value return because it is read *)
    update_tuple_expr_option s Live map
```

4.8 Other constructs

For the other constructs we do nothing, simply return the map where we save the variable met with their facts.

```
(* all other cases are ignored *)
| _ ->
    map
```

5 Examples with our analysis

Now we show one example for every statement type we regarded. For safeguard the environment, the output of the analysis has been revised and live-in and live-out table is printed side by side. Another clarification: in some examples it's possible to notice that the line numbers of the instruction are not consistent. Usually the line numbers are the original line numbers of the Ruby program. The inconsistencies are given by the RIL's transformation of the Ruby code. So, if there are instruction with the same line number, that is because it's an instruction added by the RIL's transformations.

5.1 Fundamental statements

Seq and Assignment

```
a = 1
puts a
b = 2
a = b+1
puts b
puts a
```

```
      RIL        live-in   live-out
(1) | a = 1    |         | a        |
(2) | puts(a)  | a       |          |
(3) | b = 2    |         | b        |
(4) | a = +(1) | b       | a, b     |
(5) | puts(b)  | a, b    | a        |
(6) | puts(a)  | a       |          |
```

5.2 Decisional statements

If

```
a = 1
b = 2
c = 3
if (a < b) then
  d = a + b;
  c = b * 2
else
  d = a - c
  b = 3
end
puts d
c.to_s
```

```
       RIL                  live-in              live-out
(1)  | a = 1             |                    | a                  |
(2)  | b = 2             | a                  | a, b               |
(3)  | c = 3             | a, b               | a, b, c            |
(4)  | __tmp_1 = a.<(b)  | a, b, c            | __tmp_1, a, b, c   |
(4)  | if __tmp_1 then   | __tmp_1, a, b, c   | c, d               |
(5)  |     d = a.+(b)    | a, b               | b, d               |
(6)  |     c = b.*(2)    | b, d               | c, d               |
(4)  | else              |                    |                    |
(8)  |     d = a.-(c)    | a, c               | c, d               |
(9)  |     b = 3         | c, d               | c, d               |
(4)  | end               |                    |                    |
(11) | puts(d)           | c, d               | c                  |
(12) | c.to_s()          | c                  |                    |
```

Case

```
a=4
b=2
c = 2
d = 4
```

```
case b
  when c then
    b = 3;
    a = a + 1;
    c = d - b
  when d then
    a.to_s
  else
    a = 3;
    b = 9;
    c = 10
end
a.to_s
b.to_s
c.to_s
```

```
       RIL             live-in       live-out
(1)  | a = 4         |             | a           |
(2)  | b = 2         | a           | a, b        |
(3)  | c = 2         | a, b        | a, b, c     |
(4)  | d = 4         | a, b, c     | a, b, c, d  |
(5)  | case b        | a, b, c, d  | a, b, c     |
(5)  | when c then   | a, c, d     | a, b, c     |
(7)  |    b = 3      | a, d        | a, b, d     |
(8)  |    a = a.+(1) | a, b, d     | a, b, d     |
(9)  |    c = d.-(b) | a, b, d     | a, b, c     |
(5)  | when d then   | a, b, c, d  | a, b, c, d  |
(11) |    a.to_s()   | a, b, c     | a, b, c     |
(5)  | else          |             | a, b, c     |
(13) |    a = 3      |             | a           |
(14) |    b = 9      | a           | a, b        |
(15) |    c = 10     | a, b        | a, b, c     |
(5)  | end           |             |             |
(17) | a.to_s()      | a, b, c     | b, c        |
(18) | b.to_s()      | b, c        | c           |
(19) | c.to_s()      | c           |             |
```

5.3 Loop statements

For

```
a = 1
b = 2
c = 3
d = 4
for i,j in [[a,b],[c,d],[1,2]] do
  a = i + j
  puts a
end
```

```
      RIL                                         live-in            live-out
(1) | a = 1                                      |                  | a,             |
(2) | b = 2                                      | a,               | a, b,          |
(3) | c = 3                                      | a, b,            | a, b, c,       |
(4) | d = 4                                      | a, b, c,         | a, b, c, d,    |
(5) | for j i in [[a, b], [c, d], [1, 2]] do     | a, b, c, d,      |                |
(6) |    a = i.+(j)                              | b, c, d, i, j,   | a, b, c, d,    |
(7) |    puts(a)                                 | a, b, c, d,      | a, b, c, d,    |
(5) | end                                        |                  |                |
```

While

```
a=1
guard = 7
while guard < 10 do
  a.to_s
  a = a + 1
end
a.to_s
guard.to_s
```

```
      RIL                           live-in             live-out
(1) | a = 1                       |                   | a                  |
(2) | guard = 7                   | a                 | a, guard           |
(3) | while true                  | a, guard          | a, guard           |
(3) |    __tmp_1 = guard.<(10)    | a, guard          | __tmp_1, a, guard  |
(3) |    if __tmp_1 then          | __tmp_1, a, guard | a, guard           |
(4) |       a.to_s()              | a, guard          | a, guard           |
(5) |       a = a.+(1)            | a, guard          | a, guard           |
(3) |    else                     |                   |                    |
(3) |       break                 | a, guard          | a, guard           |
(3) |    end                      |                   |                    |
(3) | end                         |                   |                    |
(7) | a.to_s()                    | a, guard          | guard              |
(8) | guard.to_s()                | guard             |                    |
```

5.4 Special statements

Method definition and MethodCall

```
def euclide(a, b)
    while(b!=0) do
        a,b = b,a%b
    end
    return a
end

i = 30
j = 56
k = euclide(i,j)
puts k
```

```
Method definition analysis:
      RIL                           live-in         live-out
(2) | while true                  | a, b          | a             |
(2) |    __tmp_1 = b.==(0)        | a, b          | __tmp_1, a, b |
(2) |    if __tmp_1 then          | __tmp_1, a, b | __tmp_2, a, b |
(2) |       __tmp_2 = false       | a, b          | __tmp_2, a, b |
(2) |    else                     |               |               |
(2) |       __tmp_2 = true        | a, b          | __tmp_2, a, b |
(2) |    end                      |               |               |
(2) |    if __tmp_2 then          | __tmp_2, a, b | a, b          |
(3) |       __tmp_3 = a.%(b)      | a, b          | __tmp_3, b    |
(3) |       (a, b) = b, __tmp_3   | __tmp_3, b    | a, b          |
(2) |    else                     |               |               |
(2) |       break                 | a, b          | a, b          |
(2) |    end                      |               |               |
(2) | end                         |               |               |
(5) | return a                    | a             |               |

MethodCall analysis:
       RIL                         live-in         live-out
(8)  | i = 30                     |               | i               |
(9)  | j = 56                     | i             | i, j            |
(10) | k = euclide(i, j)          | i, j          | k               |
(11) | puts(k)                    | k             |                 |
```

Yield and Return

```ruby
def fun
  a = 1
  b = yield a+1
  puts b
  return b + a
end
```

```
      RIL                    live-in       live-out
(2) | a = 1                 |             | a             |
(3) | __tmp_1 = a.+(1)      | a           | __tmp_1, a    |
(3) | b = yield(__tmp_1)    | __tmp_1, a  | a, b          |
(4) | puts(b)               | a, b        | a, b          |
(5) | __tmp_2 = b.+(a)      | a, b        | __tmp_2       |
(5) | return __tmp_2        | __tmp_2     |               |
```

6 Conclusions

We decided to make this project since Ruby is currently one of the most used languages, and we thought that understanding its working principles could gives us a good knowledge of the language.

Analysing the Ruby language statically is not easy. We have to take care of all the different ways the user can write constructs, and analyse them all. RIL's data-flows engine is very helpful in this situation and allows us to work on a predefined and consistent language. Due to these circumstances the liveness analysis on Ruby turned out to be an interesting challenge.

Other challenges were learning the *Ruby* language and most of all learning the *OCaml* language. Since this language is functional is not commonly used. *OCaml* turns out to be probably one of the most powerful languages that we ever used, with its high-order function and the beautiful and elegant constructs for pattern matching.

In our implementation we successfully deal with the basics constructs of Ruby, and then we tried to do something more difficult analysing the method definition within a program and the `Yield` construct, which is non-trivial challenge due to its dynamic execution.

The liveness analysis is correct within the results of our tests, but it's still possible to improve it.

Further works in this direction could be the formalisation and proof of the soundness of the result presented in this report, and eventually consider other constructs that we skipped, such as the `module` definition; another possibility could be the evaluation of other dynamic constructs of Ruby, such as the `eval` construct.

A Fix-point algorithm: RIL's backwards fix-point

```
let rec fixpoint stmt =
  (* will contain what is known before each statement *)
  let in_tbl = Hashtbl.create 127 in
  (* will contain what is known after each statement *)
  let in_tbl = Hashtbl.create 127 in
 let q = Queue.create () in
 StmtSet.iter
  (fun x ->
    Queue.push x q;
    Hashtbl.add in_tbl x DFP.empty
  ) (exits stmt);

 while not (Queue.is_empty q) do
  let stmt = Queue.pop q in
  (* IN_LIST: what is known before stmt *)
  let in_list =
   StmtSet.fold
    (fun stmt acc ->
    (* we build in-list as a list of StrMap (String -> fact) of what is known after
  executing pred *)
       try (Hashtbl.find out_tbl stmt) :: acc
       with Not_found ->
          Hashtbl.add out_tbl stmt DFP.empty;
          DFP.empty :: acc
    ) stmt.succs [] in
  (* IN_FACTS: what is known after having applied join on IN_LIST *)
  let in_facts = DFP.join in_list in

  (* the current statement stmt is updated with a new value in_facts of what we know
  before stmt is processed *)
  Hashtbl.replace in_tbl stmt in_facts;

  (* NEW_FACTS: what is known after having applied transfer to stmt based on IN_FACTS *)
  let new_facts = DFP.transfer in_facts stmt in
   try
    let old_facts = Hashtbl.find out_tbl stmt in
    (* if no change happened between the first and the second time we met stmt we stop
  looking for successors *)
    if DFP.eq old_facts new_facts
    then ()
    else begin
          (* enqueue each successor x of stmt *)
     StmtSet.iter (fun x -> Queue.push x q) stmt.preds;
     Hashtbl.replace out_tbl stmt new_facts
    end
   with Not_found ->
     StmtSet.iter (fun x -> Queue.push x q) stmt.preds;
     Hashtbl.replace out_tbl stmt new_facts
 done;

 in_tbl, out_tbl
```

B Fix-point algorithm: our implementation

```
(* function for emulating the do_while behavior. It takes as input two function: one as
    guard and one as body of the loop *)
let do_while f p =
  let rec loop() =
    f();
    if p() then loop()
  in
  loop()

(* auxiliary function that calculate our backward fix-point *)
(* return the out-facts table of the stmt *)
let rec super_fixpoint stmt in_tbl out_tbl =
  (* snode contains the statement type *)
  match stmt.snode with
  (* check if it's a Seq stmt *)
  | Seq(list) ->
      (* the starting input facts are the input facts of the seq *)
      let newfacts = ref !ifacts in
      (* we reverse the list for having a backward analysis *)
      let rev_list = List.rev list in
      (* iterate over it to calculate the facts of all the stmt of the list *)
        List.iter (fun x ->
            (* replace the old in-facts with precedent stmt out-facts *)
            Hashtbl.replace in_tbl x !ifacts;
              (* calculate this stmt newfacts ( = out-facts table) and set in out-tbl *)
            newfacts := super_fixpoint x in_tbl out_tbl;
            Hashtbl.replace out_tbl x !newfacts;
          (* these facts become the in-facts for the next stmt *)
            ifacts := !newfacts
        ) rev_list;
      (* out-fact table of the stmt is exactly the out-facts table of the last stmt
    analysed *)
      !newfacts

  (* checks if the stmt has type If *)
    | If(_, t, f) ->
      (* the guard is ignored because analysed in the transfer *)

      (* set the in-facts of both branches *)
      Hashtbl.replace in_tbl t !ifacts;
      Hashtbl.replace in_tbl f !ifacts;
      (* calculate the out-facts of both branches and save them *)
      let t_facts = super_fixpoint t in_tbl out_tbl in
      let f_facts = super_fixpoint f in_tbl out_tbl in
      Hashtbl.replace out_tbl t t_facts;
      Hashtbl.replace out_tbl f f_facts;

      (* join the 2 out-table following the if semantics, plus calculate the guard *)
      DFP.transfer (DFP.join (t_facts :: (f_facts ::[]))) stmt

  (* check if the stmt is a loop *)
    | For (_, _, b)
    | While(_, b) ->
      (* we analyse only the body because the guard is analysed in the transfer *)

      (* b_facts contains what we know before analysing the for/while *)
      let b_facts = ref !ifacts in
      let old_facts = ref DFP.empty in
      (* iteration while we reach fix-point *)
      do_while (fun () ->
     (* replace in-fact with out-facts of precedent iteration *)
     Hashtbl.replace in_tbl b !b_facts;
     old_facts := !b_facts;
     (* calculate the body out-facts *)
     b_facts := super_fixpoint b in_tbl out_tbl;
     (* calculate loop out-facts *)
     b_facts := DFP.join (!ifacts :: (!b_facts ::[]));
     b_facts := DFP.transfer !b_facts stmt )
     (* check if we reached fix-point *)
     (fun () -> not (DFP.eq !old_facts !b_facts)
    );

    (* save out-facts of the body *)
     Hashtbl.replace out_tbl b !b_facts;

     DFP.join (!ifacts :: (!b_facts ::[])) (* returns a StrMap *)

   (* if stmt is a case statement *)
   | Case (b) ->
    (* st is the list of all branch in reverse order *)
    let default = b.case_else in
    let st = match default with
     | None -> []
     | Some s -> s::[]
    in
    let st = List.fold_right ( fun (_, s) acc -> s::acc ) b.case_whens st in

    (* finalfacts will contain for each when's stmt what we know after having analysed it *)
    let finalfacts =
    (* x is each stmt in each branch of the case *)
    List.fold_left ( fun acc x ->
     (* set the in-facts of each stmt with what we know before the case stmt *)
     Hashtbl.replace in_tbl x !ifacts;
     (* calculate the out-facts table of the when stmt and save it in out_table*)
     let newfacts = super_fixpoint x in_tbl out_tbl in
     Hashtbl.replace out_tbl x newfacts;
     (* accumulate all the out-facts *)
     newfacts :: acc
    ) [] st (* _rev *)
    in
    (* join all branch out-facts *)
    let tmp_fact = List.fold_left ( fun acc x ->
       DFP.join (acc :: (x :: []))
    ) (List.hd finalfacts) (List.tl finalfacts)
    in
    (* calculate case out-facts *)
    DFP.transfer tmp_fact stmt;

  (* for all the other cases we call the transfer function directly *)
  | _ ->
   DFP.transfer !ifacts stmt
```

References

[1] Federico De Meo, Oscar Maraia, and Fabio Signorini. *A static nil-pointer analysis for the Ruby language*. Project of Static Analysis and Protection. 2012.

[2] Michael Furr et al. “Static type inference for Ruby.” In: *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM. 2009, pp. 1859–1866.

[3] Michael Furr et al. “The ruby intermediate language.” In: *ACM Sigplan Notices*. Vol. 44. 12. ACM. 2009, pp. 89–98.

[4] X Leroy, J Vouillon, and D Doligez. “The Objective Caml system, 1996.” In: *Software and documentation available on the web at http://pauillac. inria. fr/ocaml* ().