CoJaq: a hierarchical view on the Java bytecode formalised in Coq*

Patryk Czarnik, Jacek Chrząszcz, and Aleksy Schubert
{czarnik,chrzaszcz,alx}@mimuw.edu.pl

Institute of Informatics, University of Warsaw, Poland

Abstract. One of the biggest obstacles in the formalisation of the Java bytecode is that the language consists of around 200 instructions. However, a rigorous handling of metatheoretic properties of a programming language requires a formalism which is compact in size. Therefore, the actual Java bytecode instruction set is never used in the context. Instead, the existing formalisations usually cover a 'representative' set of instructions. This paper describes a design of formalisation that provides a concise set of abstract, generic instructions that can be specialised to obtain any particular bytecode instruction. In this way one can work with a manageable set of instructions to prove general facts about the Java bytecode, but at the same time all the bytecode instructions are available to enable direct verification of actual bytecode programs. A considerable part of the design has been realised in Coq.

1 Introduction

There are many tools to manipulate programs depending on their semantic properties (e.g. program translators, code refactoring tools, code optimisers) but their trustworthiness is usually based on producer's reputation. In order to guarantee correctness of such a tool one needs a formalisation of the semantics of the programming language the tool works with.

In this paper we provide a detailed and in a considerable part realised design of a formalisation of the Java Virtual Machine language (JVML) semantics. The key motivation for this project, called CoJaq[1], is to create a platform where the following two activities can be carried out. On the one hand, real programs can be translated to it and then their properties can be unequivocally expressed and proved. On the other hand, metatheoretical properties of the language can be expressed and proved. This approach has the advantage that the metatheoretical properties are proved for exactly the same language in which the actual programs are verified. In this way, the need for modelling of the language features, that may lead to inaccuracies or may be impaired by inadequate treatment of the subject issue, is eliminated.

* This work was partly supported by Polish government grant N N206 493138.

[1] Available at `http://cojaq.mimuw.edu.pl`

Formalisation of a real-world programming language is rarely taken up as this is a difficult task. In particular (a) the informal language descriptions span over hundreds of pages and it is easy to omit some subtle details that are surrounded with considerable amount of pragmatic details that have little to do with the actual semantic behaviour; (b) important semantic properties of the language are sometimes expressed implicitly and should be inferred from pieces of information spread over many pages; (c) the descriptions in natural language are not formal and therefore often ambiguous; (d) formalisation of such large entities easily goes beyond human comprehension so a non-trivial structuralisation effort is needed to achieve the final result; and (e) in case interactive theorem provers are used as support, their limitations can be reached. In fact, such a formalisation effort can be viewed as an implementation of an interpreter for the subject language in the target language of the formalisation, being either set theory, arithmetic or HOL, Coq etc. logic. As a result formalisations require significant effort very similar to the effort of programmers.

Practical languages embody many features resulting from the need for flexibility in program development. Therefore, many their constructs are reducible to a small sublanguage through a translation. Then the restricted subset can be given a fully formal description. This methodology was followed in specifications e.g. for SML [16] and JML [22,13]. In our work we take a different approach. It is hardly possible to find a strong sublanguage of the JVML that can express all the necessary features. Still, the complexity of the language is high. Therefore, we defined abstractions of the bytecode instructions following the design in [4] and then formalised the abstractions in Coq [6]. These abstractions are hierarchical and their lower levels correspond more closely to the actual instructions so that the lowest levels fit the real bytecode mnemonics in a very direct way.

In case abstraction is done for a particular phenomenon it is important to understand the principle that rules the effort. In our attempt we focused attention on runtime structures of Java Virtual Machine (JVM). The runtime structures govern the computation that is carried out by the machine and they include, among others, the heap, the set of threads and for each of them the method frame stack, the currently executed method, the program pointer in the method code etc. Different bytecode instructions operate on the runtime structures in a different way. Still, most instructions manipulate only some of the structures (e.g. the integer addition instruction manipulates only the local operand stack) and many work according to the same scheme (e.g. all conditional instructions test the top of the operand stack and change the program pointer according to the value). This approach makes it possible to discharge easily parts of proofs for metatheoretical properties that do not involve some of the runtime structures.

The key achievements of the presented formalisation are:

– The Java bytecotde instruction set has been modelled in Coq. The formalisation groups the instructions according to their handling of the JVML runtime structures. This creates a platform that can be used at the same time to verify programs and to make feasible metatheoretical proofs for the language.

– The semantics is extensive and detailed — although it is not complete, it covers a significant number of instructions and contains simplified formalisation for all aspects of the JVML in such a way that it can be extended to cover full functionality in future versions.
– A static semantic based upon types of values is developed for the instructions that use only frame runtime structures (i.e. structures such as operand stack and local variables table, but not heap). This static semantics is proved to be sound and complete with regard to the dynamic one.
– A general theorem that programs for which Hoare-style logic rules apply at each step are partially correct is proved for the above mentioned subset.

The paper is structured as follows. Section 2 describes the key design features of our formalisation. Section 3 presents the limitations of the currently existing formalisation. In Section 4 we report the related work and we conclude in Section 5.

2 Key Ideas

A formalisation of an industry standard specification for a programming language is a resource intensive task so it should be done with a particular set of requirements in mind. Here is a summary of the requirements we worked with during the formalisation of CoJaq and their impact on the design.

Multiple specification interpretations The formalisation should take into account the possibility to interpret the original specification in different yet plausible ways. The actual implementations of the standard may differ in their operation since they have taken a different approach on the implementation of a particular notion described in the specification (e.g. they use a different scheduler which has the impact on the way programs are executed). The formalisation we want to obtain should be such that it is possible to prove correct statements that describe the operation of all plausible implementations.

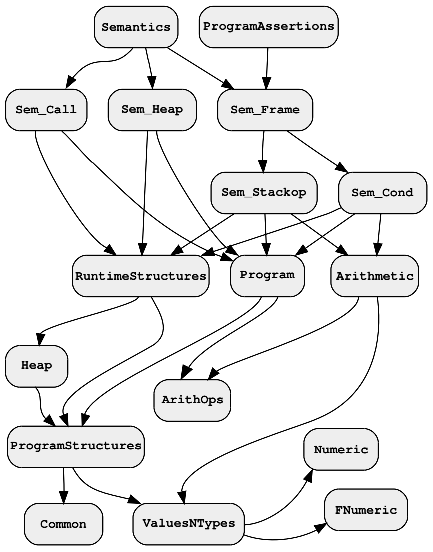


Fig. 1. Module dependencies in CoJaq. Rounded rectangles represent modules and solid arrows the relation *depends on*.

To obtain this we use three main techniques. First, the whole semantics is written in the small-step fashion. The small-step fashion is more appropriate since the natural language semantics is also formulated in the small-step fashion. Moreover, the formulation of many metatheoretic properties (e.g. immutability, purity etc.) is easier in this style. Second, the semantics is formalised as a relation. In this way the semantics we propose can be non-deterministic in places where the description is, while this would be impossible in case the formulation in the form of function was chosen. Third, we use extensively the module system of Coq to separate different aspects of the virtual machine (see Fig. 1).

Manageable set of instructions In case a metatheoretical property should be proved for a system like this, one needs to make many proofs by induction over the structure of the language. When the language is big one must consider an excessive number of cases to obtain even a very simple property. This difficulty can be avoided when similar instructions are grouped together and managed in a hierarchical structure. With such a layout many cases can be discharged on higher levels of the hierarchy resulting in smaller proofs. We took this approach in our formalisation and present it in more detail below.

Static semantics The correct operation of the semantics in the JVML strongly relies on the assumption that bytecode instructions have arguments of appropriate types. Therefore, the operation of each instruction is accompanied by a careful description of the types for its input and results. In our formalisation we separated the description of the types from the description of the actual operation of the instructions and provide two different relations for the two aspects. We believe that many proofs can be simplified due to the choice since they will not have to manage the typing information.

A logic for verification Since the formalisation should serve as a basis for verification of real programs we need a logic of programs to enable this. Therefore, a part of the formalisation must be such a logic.

2.1 Hierarchy of Instructions

Although the number of bytecode instructions is very large, one can see that many instructions are similar to one another. We can distinguish the following simple situations contributing to the proliferation of instructions.

1. A set of instructions performing the same operation for different data types. This is the case of `iload`, `fload`, `aload`, and so on. Each instruction loads a local variable value and pushes it on the operand stack, but a single instruction is applicable only to a particular type of values (`int`, `float`, and `ref`, respectively).
2. "Shorthand instructions" are defined for some most widely used argument values, as predicted by the JVML designers. An example could be the set of instructions `iload_0`, `iload_1`, `iload_2`, `iload_3`.
3. Instructions related to arithmetic and comparison often behave in similar way and differ only in an arithmetic operator. For example, `iadd`, `isub`, `imul`, and `idiv` all perform binary arithmetic operations on integers. They all pop two operands from the stack and push back one resulting value.

Each of the aforementioned cases can be simply "compressed" back to a single parametrised instruction. The examples mentioned in cases 1 and 2 can be covered by an abstract *load* instruction parametrised by a data kind (e.g. `int`) and a variable number. This one abstract instruction covers 25 JVML instructions. The case 3 can be factorised into an abstract instruction, parametrised by a data kind and a function on that kind, i.e. an abstract *binop* instruction with parameters such as addition, subtraction etc. This factorisation idea was used e.g. in Bicolano [18], where the number of instructions was reduced by almost 40%.

In [4] we decided to go one step further and factorise instructions according to runtime structures they operate on. We divided the JVML instructions into twelve parametrised abstract instructions. In the current paper, we refine this approach by organising the instructions into a hierarchy that is also followed by the definition of operational semantics. The hierarchy is presented in Fig. 2. It is represented in Coq as a number of (non-recursive) inductive types. The topmost one is `TInstruction` with 6 constructors: `I_Throw`, `I_Monitor`, `I_Invoke`, `I_Return`, `I_Heap`, and `I_Frame`. The first four represent instructions with specific access to JVM data. The fourth one, `I_Heap`, represents instructions which operate on the object heap without modifying the call stack (all variants of get, put, new and array access). The fifth one, `I_Frame`, represents the largest family of instructions which operate on the data in the method frame located at the top of the method call stack of a thread. Let us present the latter category in detail.

The constructor `I_Frame` takes one argument, which is an element of type `TFrameInstr` with 5 constructors: `FI_Load`, `FI_Store`, `FI_Inc`, `FI_Stackop`, and `FI_Cond`. The first one represents instructions which read the local variable table and write values to the operand stack. The second represents instructions which pop values from the operand stack and write the local variables array. The third one represents the only JVM instruction `iinc` which reads and writes only the variable table. The fourth one, `FI_Stackop`, represents the instructions which modify the operand stack, including all arithmetical operations. The fifth one, `FI_Cond`, represents the instructions which use the operand stack and possibly change the program counter in a different way than just moving to the next instruction. This constructor, again, takes an argument which is an inductive type representing one of the different ways a branching instruction works: unconditional jump, comparison of one element from the stack with 0, comparison of two elements from the stack, etc.

For example, the JVML instruction `if_icmpgt 26` is represented by

```
I_Frame (FI_Cond (CI_Cmp KInt CmpOp_gt offset26))
```

where `offset26` is the index of the instruction pointed by the address `26` above.

2.2 Hierarchical Definition of Semantic

Our formalisation of operational semantics follows the hierarchy of instructions. It can be seen already in a "big picture" view of Coq modules, Fig. 1, where the structure of modules implementing semantics (those with the `Sem` prefix)

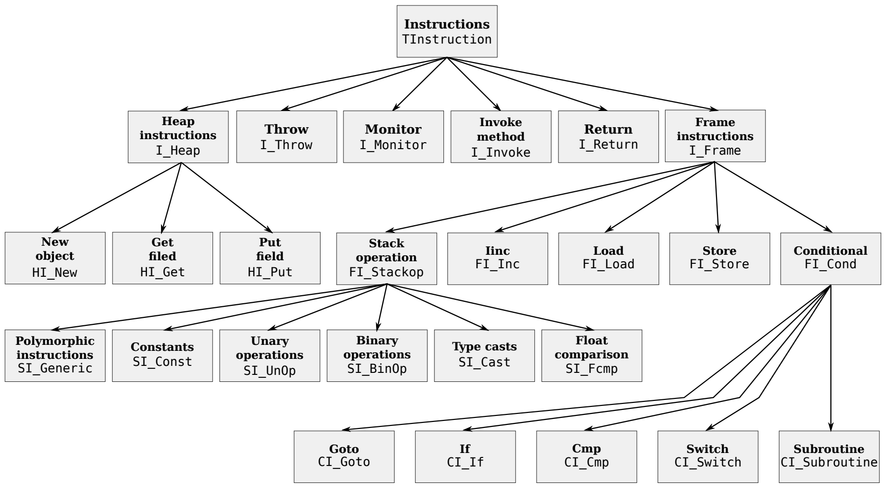


Fig. 2. Hierarchy of instruction abstractions

resembles the hierarchy of instruction representation. The semantic hierarchy is designed in such a way that the relations defining semantics of abstract instructions which are lower in the instruction hierarchy operate on a smaller fragment on the JVM state — exactly the one which is accessed by the real instructions represented by the abstract one. Due to this choice, one can handle a whole big instruction set when the property to be proved is not dependent on the details of the operation on the structures involved.

Let us present and explain the hierarchy of semantic relations on the example of `if_icmpgt 26` instruction, whose `CoJaq` representation is given above.

The small-step semantic relation is implemented in the module `Semantics` as a Coq inductive relation `step` of type `TProgram` → `TJVM` → `TJVM` → `Prop`. The relation states whether for a given program and virtual machine state a transition to another state is possible. One of its cases covers the instructions from the `I_Frame` family:

```
Inductive step (p: TProgram): TJVM → TJVM → Prop:=
| Step_frame: forall mn code finstr jvm1 th1 fr1 jvm2 th2 fr2,
   selected_thread jvm1 th1 →
   one_thread_changed th1 th2 jvm1 jvm2 →
   top_frame_changed (mn, fr1) (mn, fr2) th1 th2 →
   getMethodBodyFromProgram p mn = Some code →
   getInstruction code (frameGetPC fr1) = Some (I_Frame finstr) →
   M_Sem_Frame.semFrame code finstr fr1 fr2 →
     step p jvm1 jvm2
|  ...
```

Let us describe briefly the preconditions which are expressed with help of a number of auxiliary relations:

`selected_thread jvm1 th1` — means that `th1` is the thread selected by the scheduler in the JVM state `jvm1`;

`one_thread_changed th1 th2 jvm1 jvm2` — the JVM state `jvm2` is `jvm1` with its thread `th1` replaced by `th2`;

`top_frame_changed (mn, fr1) (mn, fr2) th1 th2` — the thread `th2` is `th1` with its topmost frame on the call stack `(mn, fr1)` changed to `(mn, fr2)`, i.e. the topmost method name (signature) `mn` did not change but the frame data `fr1` was replaced by `fr2`;

`getMethodBodyFromProgram p mn = Some code` — the method `mn` is defined in the program `p` (in particular `m` is not abstract) and its code is `code`;

`getInstruction code (frameGetPC fr1) = Some (I_Frame finstr)` — the current instruction of the current frame is of type `I_Frame`;

`M_Sem_Frame.semFrame code finstr fr1 fr2` — the specialised semantic relation of the frame instruction holds between the "old" frame `fr1` and the new one `fr2`. This relation is described below.

The relation `semFrame` specifies the semantics of `I_Frame` instructions. It has type `TFrameInstr → TFrame → TFrame → Prop` which means that, given details of an `I_Frame` instruction it is a relation on frame data and not on full JVM states. It is again an inductive relation with branches determined by the (sub)type of a given frame instruction.

Since our example instruction is a conditional jump instruction `FI_Cond` the suitable branch `StepFrame_cond` applies:

```
Inductive semFrame (code: TCode): TFrameInstr → TFrame → TFrame → Prop:=
  | StepFrame_cond: forall op pc1 vars sk1 lv1 pc2 sk2 lv2 off_op,
    M_Sem_Cond.semCond op lv1 lv2 off_op →
    stackTopValues lv1 lv2 sk1 sk2 →
    pc2 = calculate_pc off_op code pc1 →
    semFrame code (FI_Cond op) (mkFrame vars sk1 pc1) (mkFrame vars sk2 pc2)
  | ...
```

Note that the number of premises here is much smaller than in the definition of the general `step` relation. Actually, there is only one structural premise, `stackTopValues lv1 lv2 sk1 sk2`, which says that the operand stack `sk2` is `sk1` with its topmost values `lv1` replaced by `lv2`. Apart from this, `M_Sem_Cond.semCond op lv1 lv2 off_op` says that the lower level semantics for `FI_Cond` instructions permit the transformation of values at the top of the stack from `lv1` to `lv2` and optionally generates a jump to `off_op`. Lastly, the equation for `pc2` calculates the proper next instruction according to `off_op` and the current position. The final line of the above `semFrame` branch shows that evaluation of the `FI_Cond` instruction does not change local variables.

The precise semantics of the `FI_Cond` abstract instruction does not interact with the whole frame, but only with the parts it really needs, i.e. the top values from the stack. And the actual effect of the instruction on the values taken from the operand stack is implemented in the `semCond` relation of type `TCondInstr → list TValue → list TValue → option TOffset → Prop`, which given conditional instruction details should be understood as a partial function from a list of values (popped from the operand stack) to the list of values (to be pushed back on the operand stack) and optional jump address. It is realised as an inductive relation with branches determined by the conditional instruction details `TCondInstr`. Of course the concrete comparison of integers is done in the proper arithmetic module.[2]

The hierarchical structure of semantics has at least three advantages. First of all, it prevents code duplication, as otherwise the `step` relation e.g. for all the `I_Frame` instructions would have almost identical premises corresponding to extracting the suitable fragment of the JVM state. Second, when proving some properties of the semantics, the necessarily large proof is also hierarchically organised into lemmas and therefore easier to manage than a big monolithic one. Moreover, if the property at hand is not relevant for a large part of instructions, chances are that many of the irrelevant instructions will be discharged at a high level of semantic hierarchy, e.g. one would discharge the whole `I_Frame` branch of the `step` relation and not many separate instructions one by one. The third advantage is the possibility to develop some proof techniques like VCGen, Hoare logic etc. only for fragments of the semantics, if the whole semantics it too complex to cover. The hierarchical structure of the semantics provides again, a natural delineation of fragments to do and to ignore.

2.3 Static Semantics

The dynamic semantics defines the actual behaviour of bytecode programs. Still, a program can run correctly, according to the JVM specification, only provided that it is type correct. Therefore, one needs a description of how execution of instructions modifies the types of the elements stored in the program runtime structures. This is a basic step to prove correctness of the JVML verification procedure. In our formalisation we accompany definitions of the dynamic steps with static ones. In particular, the conditional instruction operation defined through

```
semCond: TCondInstr → list TValue → list TValue → option TOffset → Prop
```

is accompanied by the relation

```
staticSemCond: TCondInstr → list TKind → list TKind → option TOffset → Prop
```

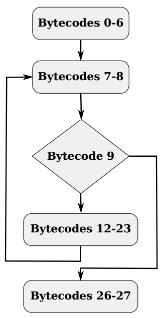
which says that the operation in question given an operand stack with top elements of types enumerated in the first list (`list TKind`) returns a stack with the top elements replaced with values of types enumerated in the second list (`list TKind`) and optionally moves the program counter given number of steps (`option TOffset`).

A crucial property of such a static semantics is that it is consistent with the dynamic one. We proved that the static semantics is sound and complete with respect to the dynamic one. First, we showed that every possible step in the dynamic semantics is covered by a corresponding step at the static semantics level. For `semCond` the fact is expressed by a property of the following form:

[2] The arithmetic module for the kind `int` was taken from the earlier work by David Pichardie in Bicolano [18].

```
public static int m() {
  int i = 0;
  int n = 50;
  int r = 0;
  while (i <= n) {
   r = i + r;
   i++;
   r = i + r;
  }
  return r; }
```

(a)



(c)

public static int m();

```
 0: iconst_0       // int i = 0;
 1: istore_0
 2: bipush 50      //int n = 50;
 4: istore_1
 5: iconst_0       //int r = 0;
 6: istore_2
 7: iload_0
 8: iload_1
 9: if_icmpgt 26   // i > n
12: iload_0
13: iload_2
14: iadd           // i + r
15: istore_2       // r = ...
16: iinc  0, 1     // i++
19: iload_0
20: iload_2
21: iadd           // i + r
22: istore_2       // r = ...
23: goto  7        //end of loop
26: iload_2        // r is returned
27: ireturn
```

(b)

Fig. 3. An example of a method code and its bytecode. (a) The source code of a method, (b) a bytecode listing of the method, and (c) the control flow graph of the bytecode.

```
forall (op: TCondInstr) (ooff: option TOffset) vl1 vl2,
  semCond op vl1 vl2 ooff →
    staticSemCond op (kindOfValues vl1) (kindOfValues vl2) ooff.
```

Second, we demonstrated that every step in the static semantics is motivated by a corresponding step in the dynamic one, which is expressed as the following property:

```
forall (op: TCondInstr) (ooff: option TOffset) ks1 ks2,
  staticSemCond op ks1 ks2 ooff →
    exists vs1 vs2, kindOfValues vs1 = ks1 ∧ kindOfValues vs2 = ks2 ∧
      semCond op vs1 vs2 ooff.
```

Currently, we provide and prove such properties for `IFrame` instructions.

2.4 Program Verification

A systematic process of JVML program verification can be performed in the following way: one writes formulas that describe the states between every two

```
Definition code: TCode := codeFromList
 [(* 0*) I_Frame (FI_Stackop (SI_Const KInt zero));
  (* 1*) I_Frame (FI_Store KInt var0);
  (* 2*) I_Frame (FI_Stackop (SI_Const KInt (VInt (Num.const n))));
  (* 3*) I_Frame (FI_Store KInt var1);
  (* 4*) I_Frame (FI_Stackop (SI_Const KInt zero));
  (* 5*) I_Frame (FI_Store KInt var2);
  (* 6*) I_Frame (FI_Load KInt var0); (* beginning of the loop *)
  (* 7*) I_Frame (FI_Load KInt var1);
  (* 8*) I_Frame (FI_Cond (CI_Cmp KInt ArithmeticOperators.CmpOp_ge
                                    (offsetFromPosition 19%nat)));
  (* 9*) I_Frame (FI_Load KInt var0);
  (*10*) I_Frame (FI_Load KInt var2);
  (*11*) I_Frame (FI_Stackop (SI_Binop KInt ArithmeticOperators.BinOp_add));
  (*12*) I_Frame (FI_Store KInt var2);
  (*13*) I_Frame (FI_Inc var0 (Num.const 1));
  (*14*) I_Frame (FI_Load KInt var0);
  (*15*) I_Frame (FI_Load KInt var2);
  (*16*) I_Frame (FI_Stackop (SI_Binop KInt ArithmeticOperators.BinOp_add));
  (*17*) I_Frame (FI_Store KInt var2);
  (*18*) I_Frame (FI_Cond (CI_Goto (offsetFromPosition 6%nat)));
  (*19*) I_Frame (FI_Load KInt var2) (* after the loop *)
  (*20*) I_Return (Some KInt) (* return *) ].
```

Fig. 4. The method code from Fig 3 translated to our formalisation

bytecode instructions and then proves that starting from a state satisfying the formula before an instruction if the semantic step of the instruction is taken then the resulting state satisfies the formula after the instruction.

Consider the small program given in Fig. 3. It consists of initial assignments of constants to local variables and the loop that basically calculates the sum of first n odd numbers, which happens to be equal to n^2. Its CoJaq counterpart is given in Fig. 4. First of all, note that labels in bytecode are positions in bytes, whereas in the Coq counterpart they are consecutive numbers. Second, the CoJaq code is parametrised by n, while in Java and JVML n is replaced by a concrete constant 50. The proof of program correctness is of course done for arbitrary (but small enough) n.

The proof process starts with proving a number of auxiliary lemmas about properties of small `int32` numbers. After that we define properties describing the state before given instructions, e.g:

```
Definition s8_prop frame :=
  pcToPosition (frameGetPC frame) = 8%nat
  ∧ exists i, exists r, stack_values frame [n; i]
      ∧ var_value frame var0 i ∧ var_value frame var1 n
      ∧ var_value frame var2 r ∧ r = i*i ∧ 0 <= i ∧ i <= n.
```

The above definition says (i) that the program counter of the current frame is at position 8, (ii) that the values on the operand stack correspond to the values of appropriate local variables, and (iii) that the abstract loop invariant is satisfied, i.e. `r=i*i`, where `i` is in the appropriate range.

Once the state properties are defined, we prove a number of lemmas about transitions, e.g.

```
Lemma trans_7_8: forall frame frame’,
  s7_prop frame → SF.stepFrame code frame frame’ → s8_prop frame’.
```

The proofs consist mostly in unfolding definitions, decomposing conjunctions and inverting inductive relations. They can be automated to a large extent.

After proving transition lemmas, one can establish that reachable program states are described by the aforementioned state properties. Hence, one can show the partial correctness of the program, i.e, when it is started in the initial state and arrives after instruction 19 then the operand stack holds n^2:

```
Theorem partial_correctness: forall frameF,
  pcToPosition (frameGetPC frameF) = 20%nat →
    SF.stepsFrame code frame0 frameF → exists res,
      frameGetLocalStack frameF = [(VInt res)] ∧ Num.toZ res = (n * n).
```

In this way we proved the desired functional property of the bytecode program in Fig. 3.

2.5 Hoare Logic

The method of the partial correctness proof presented above can be generalised in the Hoare logic style. This is done in `ProgramAssertions` module. The module is parametrised with a container of assertions associated with program points. These assertions are properties of runtime structures or, in other words, properties of the state. The intent is that they define the set of states that are desired to turn up before the instruction at the program point is executed.

This framework makes it possible to state in general terms when the partial correctness property is valid. For the property to hold we assume that at each program point the relation `step` links states that obey the assertion before and after the instruction at the point. Under this condition, the execution of any sequence of steps in the program also connects states that obey the assertions at the beginning of the sequence and at its end.

This generalisation gives an automatic way to extend the correctness proofs for particular instructions to the partial correctness of methods. Still, the most laborious part of the work is to generate appropriate assertions for particular instructions and prove them. This work cannot be encapsulated in a neat theorem and must be done anyway. The only way to simplify the work is to devise an appropriate verification condition generator and a series of Coq tactics that automate the tedious handling of all details that are necessary for formal reasoning, but do not touch the essence of the correctness proof for a particular code. The proof presented in Section 2.4 shows that most of the state properties could be generated automatically by a suitable VC-gen.

3 Missing Features in the Formalisation

The current version of `CoJaq`, as already mentioned, does not cover all the details of the language. Within the limited resources in our reach we had to choose the priorities for the formalisation of features. We worked with the following assumptions:

- The support for formal verification of simple programs with arithmetic operations and loops should be obtained quickly. As soon as this is achieved, a development of the tools to support formal treatment of programs can be started.
- Formalisation of instructions in different categories should be done to justify and test decomposition of the formalisation into different modules.
- We expected that some details of the overall design were subject to change. Therefore, we omitted formalisation of certain aspects of the language. There were two major reasons for this. Some of the aspects such as multithreading are very difficult to model so they need a well established basic structure to undertake. To facilitate the possible changes we also kept the number of actually formalised mnemonics small and avoided ones where similar functionality was already realised elsewhere (e.g. exception handling is similar to either jumps or returns, and 64-bit values are similar to 32-bit ones).

Here is the summary of the major omissions in our formal account of the JVML.

Advanced multithreading and Java memory model Full handling of multithreading requires at least three major pieces of work to be accomplished (i) axiomatisation of the thread scheduler, (ii) axiomatisation of the Java memory model, and (iii) support for native methods to make available functions such as thread creation, `wait`, or `signal`. Currently we have partial implementation of (i) and (ii). We use a simplified scheduler that non-deterministically chooses the thread for the next instruction to be executed. We also introduced the possibility to see the heap in a different way depending on the current thread and realised two strategies of heap synchronisation, namely, (a) no synchronisation between threads and (b) full synchronisation of the heap access.

Exceptions All the needed structure to implement the instructions is in place, but the mnemonics have not been described yet.

Native methods The native methods are a mechanism that makes it possible to extend ad hoc the JVML functionality in an arbitrary way. Therefore, native calls require a mechanism to add an arbitrary step `step jvm1 jvm2` relation for a native method invocation. Currently, this can be added directly in the `Sem_Call` module. However, this is not satisfactory solution since it breaks the integrity of our formalisation.

Bytecode instructions not covered so far While the shape of instructions categorisation tree is broadly established and it shows a potential to cover the whole JVML, the current version of the semantics does not handle all the instructions. The instructions not described yet at the top level of the categorisation are `I_Throw` and `I_Monitor`. At lower levels of the formalisation we miss also the formalisation of `tableswitch` and `lookupswitch`, `jsr` and `ret`, as well as arrays support for heap and the support for 64-bit instructions.

Validation against existing implementations A formalisation of a programming language requires a method to compare the formalisation with existing implementations. The current version of the platform does not have a direct method to make such a comparison. However, we designed the formalisation with one such method in mind so that it is possible to add such comparing infrastructure in the future when appropriate resources will become available.

In our design, we assumed that the comparison will be done through testing. A typical test will consist of checking if a small procedure returns a particular result given a particular input. In such case it is straightforward to generate precondition and postcondition that express such a relation. Moreover, one can compute how many bytecode instructions were executed during such a run. Based upon this number, n, we can automatically generate verification conditions that suffice for verification of the code under the restriction that no jump target is visited more than n times. Then the verification conditions should have so simple structure to admit automatic generation of their proofs. It is important to note that such proofs will have to use inversion to transform the condition after the executed instruction to the one before. This way of handling is a result of the design choice that program steps are formalised as inductive relations.

4 Related Work

A systematic reduction of a large set of JVML instructions to a small one by means of abstraction was given by Yelland [25]. He proposed a language μJVM with a modest set of instructions that transform program continuations. Next, a translation was provided for the actual bytecode instructions. In fact, one can view the work as a continuation style denotational semantics for the JVML written in Haskell, which makes it immediately modular and executable. One important advantage of the formalisation is that the Haskell type system corresponds there to type correctness verification. In our approach we formalise the language in small step fashion and the correctness proof for verification procedure needs to be done separately. Moreover, μJVM works on a different level of abstraction — instructions in CoJaq correspond in a hierarchical way to instructions in the JVML, while in the case of μJVM a translation is required.

Formal programming languages semantics on paper can be dated back at least to the semantics of Pascal [12]. The semantics of Java and the JVML was given in a notable book by Stark et al [24]. A number of formal accounts of the JVML are available in the literature. Their extensive overview can be found in the works of Hartel and Moreau [11] as well as Freund and Mitchell [10]. We present here a brief overview of those realised in mechanised frameworks.

Mechanised formalisations of the JVML Probably the earliest effort to mechanically formalise the JVML was done by Pusch [20]. She formalised the language in Isabelle/HOL by direct representation of general instructions that group bytecode operations. The language covered such aspects as low-level control flow, integer types, classes, methods, and arrays. She proved the correctness of the JVML verifier. The formalisation largely corresponds to an earlier formalisation on paper done by Qian [21], which was also formalised in Specware [5].

Bertot validated in Coq [3] the correctness of soundness proofs for the fragment of the JVML concerned with object initialisation. This work was based upon an early version of the work by Freund and Mitchell [9].

An important formalisation was proposed by Leroy [15]. This formalisation is focused on JavaCard version of JVML and offers a Coq formal proof that the JVML verifier is correct and that a preverified type information can serve to guarantee type correctness after a type checking procedure is executed.

Klein and Nipkow [14] proposed probably the most extensive work concentrated on the JVML verification. They provided a model of Java called Jinja and a formalisation of the JVM language model with 15 instructions that includes such aspects of the JVML as low-level control flow, integer numeric operations, classes, arrays, methods, exceptions, casts, and bytecode subroutines. They constructed a verified compiler of Jinja to their model of JVM as well as a JVML verifier. All the verification of the procedures was done in the theorem prover Isabelle/HOL. As a result they obtained a unified model for the source language, the virtual machine, and the compiler.

A considerable fragment (over 70 instructions) of the whole instruction set of the JVML was modelled by Pichardie [18] in Coq. The work was similar in spirit to the one of Bertelsen [2] and modelled directly the instructions. The semantics was done both in the small-step and big-step fashion and the two were proved equivalent. This was probably the most ambitious and largely successful attempt to make a formal account of the full bytecode instruction set. However, the drawback of this approach was such that the number of instructions made the formalisation unwieldy in the context of proving metatheorems for JVML e.g. that a JVML verification algorithm is correct.

Another attempt to formalise JVML was done by Atkey [1] in Coq. The most important feature of the attempt is that it uses the Coq program extraction to make possible extraction of Ocaml programs that work as a JVM. In this way it is possible to efficiently validate the operational semantics encoded in Coq against real JVMs and test if the results obtained in the two environments agree.

A recent work of Demange et al [7] can also be viewed as a formalisation of the JVML. The authors present a semantics of a chosen set of bytecode instructions in Coq and a translation of bytecode to a stackless representation to make a basis for formal analysis of bytecode compilation and its optimisation to native code in JIT or standard compilers. Moreover, a semantics in Coq is given for the target language. In this way they obtain two semantic accounts of the bytecode and they prove that they are equivalent.

An interesting exercise in formal methods was proposed by Posegga and Vogt [19]. They showed how model checking can be applied to verify functional properties of a JVML program.

Formalisations of other low level languages A number of interesting formalisations of other low-level languages showed up in last years. Sarkar et al [23] proposed a formalisation in HOL of the x86 instruction set together with a memory model of causal consistency. The goal of the formalisation was to give deeper intuition for low-level programming and sound foundation for further formal developments. Another interesting formalisation was given for ARM instruction set by Fox and Myreen [8]. The main focus of the formalisation is to make a platform to work on verification of programs in ARM assembly language. A distinguishing feature of the approach is that the platform contains a developed tool set to test it against the existing hardware.

5 Conclusions

In thinking about programs, programmers tend to focus only on chosen aspects of program execution. This often agrees with the assumption that the state of certain runtime structures that govern JVM is irrelevant for the operation of the particular instruction. We took this view and grouped the JVML instructions based upon the way they operate on the runtime structures. In this way we obtained a hierarchical decomposition of the Java instruction set and formalised in Coq a considerable part of it. We believe that in this way it will be possible to both prove metatheoretic properties of the JVML and prove correctness of particular programs. In addition to the semantics of instructions we developed a static type-based semantics to separate the reasoning concerning types from the one concerning execution and a framework for proving partial correctness of JVML programs. The whole development consist currently of over 7 KLOC of Coq files.

References

1. Atkey, R., *CoqJVM: An executable specification of the Java Virtual Machine using dependent types*, in: M. Miculan, I. Scagnetto and F. Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, LNCS **4941** (2008), pp. 18–32.
2. Bertelsen, P., *Dynamic semantics of Java bytecode*, Future Gener. Comput. Syst. **16** (2000), pp. 841–850.
3. Bertot, Y., *Formalizing a JVML verifier for initialization in a theorem prover*, in: G. Berry, H. Comon and A. Finkel, editors, *Computer Aided Verification*, LNCS **2102**, Springer-Verlag, 2001 pp. 14–24.
4. Chrząszcz, J., P. Czarnik and A. Schubert, *A dozen instructions make Java bytecode*, ENTCS **264** (2011), pp. 19–34.
5. Coglio, A., A. Goldberg and Z. Qian, *Toward a provably-correct implementation of the JVM bytecode verifier*, in: *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings* (2000), pp. 403–410, vol. 2.
6. Coq development team, *The Coq proof assistant reference manual V8.4*, Technical Report 255, INRIA, France (2012), `http://coq.inria.fr/distrib/V8.4/refman/`.
7. Demange, D., T. Jensen and D. Pichardie, *A provably correct stackless intermediate representation for Java bytecode*, in: *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, LNCS **6461** (2010), pp. 97–113.

8. Foxand, A. and M. O. Myreen, *A trustworthy monadic formalization of the ARMv7 instruction set architecture*, in: M. Kaufmann and L. C.Paulson, editors, *Interactive Theorem Proving*, LNCS **6172**, Springer-Verlag, 2010 pp. 243–258.
9. Freund, S. N. and J. C. Mitchell, *The type system for object initialization in the Java bytecode language*, ACM Trans. Program. Lang. Syst. **21** (1999), pp. 1196–1250.
10. Freund, S. N. and J. C. Mitchell, *A type system for the Java bytecode language and verifier*, J. Autom. Reason. **30** (2003), pp. 271–321.
11. Hartel, P. H. and L. Moreau, *Formalizing the safety of Java, the Java virtual machine, and Java card*, ACM Comput. Surv. **33** (2001), pp. 517–558.
12. Hoare, C. A. R., *An axiomatic definition of the programming language PASCAL*, in: *Proceedings of the International Sympoisum on Theoretical Programming* (1974), pp. 1–16.
13. Jacobs, B. and E. Poll, *A logic for the Java Modeling Language JML*, in: *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, LNCS **2029** (2001), pp. 284–299.
14. Klein, G. and T. Nipkow, *A machine-checked model for a Java-like language, virtual machine, and compiler*, ACM Trans. Program. Lang. Syst. **28** (2006), pp. 619–695.
15. Leroy, X., *Bytecode verification on Java smart cards*, Softw. Pract. Exper. **32** (2002), pp. 319–340.
16. Milner, R., R. Harper, D. MacQueen and M. Tofte, "The Definition of Standard ML – Revised," The MIT Press, 1997.
17. MOBIUS Consortium, *Deliverable 3.1: Bytecode specification language and program logic* (2006), available online from `http://mobius.inria.fr`.
18. Pichardie, D., *Bicolano – Byte Code Language in Coq* (2006), `http://mobius.inria.fr/bicolano`. Summary appears in [17].
19. Posegga, J. and H. Vogt, *Byte code verification for Java smart cards based on model checking*, in: J.-J. Quisquater, Y. Deswarte, C. Meadows and D. Gollmann, editors, *Computer Security – ESORICS 98*, LNCS **1485**, Springer-Verlag, 1998 pp. 175–190.
20. Pusch, C., *Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL*, in: R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, LNCS **1579** (1999), pp. 89–103.
21. Qian, Z., *A formal specification of Java Virtual Machine instructions for objects, methods and subrountines*, in: *Formal Syntax and Semantics of Java*, LNCS **1523** (1999), pp. 271–312.
22. Raghavan, A. D. and G. T. Leavens, *Desugaring JML method specifications*, Technical Report TR #00-03d, Iowa State University (2000–03).
23. Sarkar, S., P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen and J. Alglave, *The semantics of x86-CC multiprocessor machine code*, in: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2009), pp. 379–391.
24. Stärk, R. F., J. Schmid and E. Börger, "Java and the Java Virtual Machine: Definition, Verification, Validation," Springer, 2001.
25. Yelland, P. M., *A compositional account of the Java virtual machine*, in: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1999), pp. 57–69.