An approach to Subroutine Elimination

Leonardo Rodríguez Viacava
lrodrigu@fing.edu.uy

INCO – PEDECIBA
Facultad de Ingeniería
Universidad de la República
Uruguay

Project Lemme
Sophia Antipolis
INRIA
France

Abstract

Subroutines seem to be more a problem than a solution for the Byte Code Verifier's world, especially with resource constrained devices like Java Card. The elimination of subroutines form the Java bytecode would allow the construction of more efficient and precise Byte Code Verifiers. Here we specify a transformation for eliminating subroutines and we prove that its preserves the semantics of the Java program been transformed. Al this is done on top of the COQ Proof Assistant.

Keywords : Byte Code Verifiers, subroutines, Java Card, COQ, formal proofs

1. Introduction

Since its beginnings Java was a well suited programming language for development of Web applications using web applets, and later also for distributed applications, Java Card applications and mobile applications, among others. All this applications take full advantage of the possibility of downloading untrusted code from different sources and executing it without endangering the environment. As always, this feature is not for free, it introduces serious security issues that must be solved out.

The Byte Code Verifier (BCV) [2] is one of the most crucial components used by the Java Virtual Machine (JVM) to address this security issue. The objective of the BCV is to verify that certain properties, e.g. type correctness, are fulfilled by the applications before the execution, avoiding the execution of malicious code that might try to perform an attack. Performance and correctness are qualities of major importance for and BCV algorithm, since an inefficient algorithm would degrade the general performance of the VM, and a bug in it could lead to dangerous security issues.

As it is documented in the JVM specification [2] the subroutines of the JVM Language (JVML) are a major source of complexity for the BCV algorithms. In order to resolve the difficulties introduced by the subroutines, BCV algorithms should increase its complexity, reduce its performance and also reduce its precision. As studied in [1] the advantages of using subroutines might not be worth the degradation in the performance and precision that it introduces. One alternative to solve these problems would be eliminating subroutines from the bytecode. Subroutines can be easily using other JVML instructions. New virtual machines like the K Virtual Machine (KVM) [7] which is a Java virtual machine especially designed for small, resource-constrained devices such as cellular phones, pagers, PDAs, and so forth; already don't support subroutines, so some kind of subroutine elimination transformation should take place before loading the Java code intro the VM.

This paper addresses this need by proposing a simple subroutine elimination transformation that could be applied to any Java program to eliminate all subroutines occurrences form it. In addition to this it will also formally prove that this transformation preserve the semantics of the Java program for a defined JVML instructions subset. By doing so both BCV's desired qualities can be achieved, performance because the elimination of subroutines allow the construction of BCV more efficient and simple; and correctness since a formal proof of correctness is done for this transformation.

The formal platform used to specify the transformation and to perform the proofs is COQ Proof Assistant [10, 11]. The transformation is implemented and tested over a formal executable specification of the Java Card Virtual Machine (JCVM) that is implemented in COQ. This implementation was done by the Certicartes Project [8]. This work has not been completed yet, so this paper presents its current state.

The remaining sections of the paper are organized as follows. In Section 2 we present a general overview of Byte Code Verifiers. Section 3 introduces the concept of subroutines in the Java bytecode, has a discussion of the advantages and disadvantages of using them, and presents an intuitive idea of the elimination procedure to use. Section 4 describes the platform used for the implementation and describes the actual transformation. Section 5 presents the proof to be done in order to ensure the correctness of the transformation. Finally, Sections 6 and 7 conclude and introduce future work to be done in this area.

2. Overview of Byte Code Verifiers

2.1 Introduction to Byte Code Verifiers

The Byte Code Verifier is one of the components used by the JVM to enforce its security. This is done by performing a static analysis of the bytecode before its execution, assuring that it fulfills certain properties. The most important properties that must be checked by the verifier are: type correctness (Java bytecode is a typed language, so it guarantees that all instructions receive arguments of the correct types), stack overflow and underflow, program counter bounds (the program counter must be kept inside the methods bytecode) and object initialization (when a new object is created, one of the constructors of the object must be called before the object can be used) [2, 4].

A BCV algorithm must procure two fundamental qualities, correctness and performance. The algorithm must be correct because it is meant to enforce security in the JVM and also it may be used in platforms like Java Card where security is a major issue. A bug in the BCV may lead to many new security issues exploitable by malicious bytecode. Special efforts should be expended in assuring its correctness. Several formalization efforts have been made to achieve this goal, for example [3, 5, 15].

The algorithm must also be efficient, since the verification is a process that every application must go through before it can be executed. An inefficient algorithm would degrade the general performance of the virtual machine. Another argument is that the algorithm must be able to perform verification in low resources devices like Java Card. The verification process is done by performing a static analysis of the application's class files. Doing the verification statically is fundamental to achieve performance because the bytecode verification takes place only one time when the class is been loaded (before its execution).

Although the JVM specification [2] state that the BCV must be independent of the compiler and must be capable of verifying code forma any actual or future compiler, currents BCV implementations don't fulfill this requirement. They do this to obtain better performances for the verification algorithm.

Intuitively in order to type check a class the BCV algorithm should analyze the bytecode of each class's method and trace all possible execution paths checking that for each step of execution the bytecode is well typed. In practice analyzing each execution path independently is not feasible because the time and resources needed to do it. Actual BCV algorithms tend to assume some restrictions for the bytecode and perform some approximations during the analysis of the bytecode in order to improve performance. As a consequence, the BCV algorithm some times rejects correct bytecode, so the precision of the algorithm is reduced.

Different techniques have been used to perform the verification. The first verification algorithm was proposed by Sun [2], and it combines techniques of abstract interpretation [6] and dataflow analysis [6] to perform the verification. Abstract interpretation is used to perform an abstract execution of the bytecode only considering the types of the local variables and stack values. In order to deal with branches, a dataflow analysis technique is used, allowing analyzing each possible execution flow. The algorithm proposed by Sun performs a mono-variant analysis, maintaining one state per instruction in the bytecode [2, 4]. Having only one state per instruction implies that for instructions that are target of more than one execution path, the states of all paths should be merged in one state. This merging some times can lead to a loss of precision in the analysis. The mono-variant analysis has the advantage that is simple, consumes few resources and has a reasonable performance.

2.2 Subroutines

The concept of subroutine came up to the Java world to allow compilers the generation of size optimized bytecode. A subroutine is a set of instructions included in the method's bytecode that can be called from any part of the method and share the same activation record with its caller. They are present only at the bytecode level and they are completely transparent to the Java Source. In the JVML there are two instructions designated to manage the subroutines, the `jsr` (Jump-SubRoutine) instruction that allow the call of a subroutine and the `ret` instruction that is used to return from a subroutine.

The main use of the subroutines is to implement the `try-catch-finally` block of Java [2], because the semantic of the `finally` demands that its code must be the last code executed of the `try-catch-finally` block. For example, as we can see in Figure 2.1, the `finally` should be called from line 9 after the normal (without unhandled exceptions) termination of the try block, it should be called from line 18 if the exception is raised and it should be called from line 21 if an unhandled exception is raised. For a more detailed explanation see section 7.13 of [2].

Java Source

```
public class Test{
  private int i = 0;
  public void example1() {
    try{
      i++;
    }catch (Exception e){
      i++;
    }finally{
      i--;
    }
  }
}
```

Java Compiled Bytecode

```
1  public void example1() {
2     int lv1, lv3, lv3;
3     L0 aload_0  //push reference from lv0
4        dup      //duplicate top stack word
5        getfield Test.i //push contents of field Test.i
6        iconst_1 //push 1 onto stack
7        iadd     //add the two ints on the stack
8        putfield Test.i //set field Test.i
9     L1 jsr L4   //push address of next opcode then goto L4
10       goto L5  //goto L5
11    L2 astore_1 //pop reference into lv1
12       aload_0  //push reference from lv0
13       dup      //duplicate top stack word
14       getfield Test.i //push contents of field Test.i
15       iconst_1 //push 1 onto stack
16       iadd     //add the two ints on the stack
17       putfield Test.i //set field Test.i
18       jsr L4   //push address of next opcode then goto L4
19       goto L5  //goto L5
20    L3 astore_2 //pop reference into lv2
21       jsr L4   //push address of next opcode then goto L4
22       aload_2  //push reference from lv2
23       athrow   //throw reference currently on stack
24    L4 astore_3 //pop reference into lv3
25       aload_0  //push reference from lv0
26       dup      //duplicate top stack word
27       getfield Test.i //push contents of field Test.i
28       iconst_1 //push 1 onto stack
29       isub     //subtract int at stack top from int below it
30       putfield Test.i //set field Test.i
31       ret 3    //return from subroutine. Use address in lv3
32    L5 return   //return void from method
33    catch java/lang/Exception
34       Start offset(L0), End offset(L1), Handler offset(L2)
35    catch ANY (finally)
36       Start offset(L0), End offset(L3), Handler offset(L3)
37 }
```

Figure 2.1: try-catch-finally compilation example

The subroutines can also be useful to the compilation of the synchronized statements, in order to avoid the repetition of the code that frees the locks before leaving the synchronized block, for an example see [1]. But this approach is not in the JVM specification and is not widely use by compilers.

2.3 Subroutines and BCVs

The mono-variant technique proposed by Sun to perform the bytecode verification has problems handling bytecode that includes subroutines. By definition a subroutine can be called from any part of the method, so a mono-variant algorithm would merge the states of all execution paths at the first instruction of the subroutine. This merge usually results in a major loss of precision, which mainly affect variables not used by the subroutine (this problem is explained in [1, 2, 3, 4, 9]). A direct consequence of this is the rejection of much of the bytecode that includes subroutines even if this bytecode is correct. Solving this problem considerably increase the complexity of the verification algorithm. Sun presents a solution for this problem in its proposal [2].

This problem could also be treated using poly-variant analysis. Using this technique an independent state can be maintained for each subroutine call, avoiding the merge of states that cause the loss of precision. This approach was used to implement the Off-Card Bytecode Verifier of the Sun Java Card Development Kit. This technique can solve the subroutines problem by obtaining higher precision, but it needs more resources and tends to be less efficient than the Sun algorithm. This technique is too expensive to be preformed inside a resource constraint device like a Java Card.

Model checking of abstract interpretation has also been used to perform the bytecode verification [12, 13]. It implements in some way the intuitive idea presented before (with poly-variant), checking all possible states reached when executing a method. A technique like this doesn't present any problem with subroutines or with branches, but it is a very expensive process. Some research efforts have been made in these directions [12] and some interesting results have been obtained, but the cost of using it is still too high.

3. Subroutine Elimination

3.1 Advantages and disadvantages of subroutines

The subroutines where introduced in JVML in order to optimize the size of the bytecode generated by compilers. They allow the encapsulation of method's repetitive bytecode in a subroutine and call it from anywhere it is needed to do so (always from within the method's bytecode). But nothing comes for free, as we previously presented using subroutines introduce some difficulties for BCVs. The studies made in [1] about the advantages and disadvantages of having subroutines, suggests reconsidering the use of subroutines. It may not be worthy using them in all platforms.

Subroutines increase the complexity of the algorithm that performs the bytecode verification. The simple algorithm proposed by Sun in section 4.9.2 [2] would try to handle the subroutine call as a simple branch in the bytecode. The result is merging the states of all execution paths that call the subroutine. This merge take place at the first instruction of the subroutine. Potentially the type of all local variables that are not used by the subroutine are different from one state to the others, so the result of merging those states may have an undefined value in those variables. This imprecision in the approximation would be carried through the subroutine body by the analysis algorithm and when it reaches the return point of the subroutine would be carried also to the instruction after each call to the subroutine.

This loss of precision is unacceptable, and ends up in the rejection of most of the correct bytecode that includes subroutines. This problem with the subroutines and local variables is commonly presented saying that the subroutines are polymorphic over the local variables that they do not use. Sun presented an informal solution to this problem in section 4.9.6 of [2], but there are also some other solutions proposed in [3, 5, 6].

This increase in complexity came with an increase in the time and space needed by the verification algorithm, and as a consequence the general performance of the JVM suffers degradation. This problem became more important when working with embedded devices where resources are very limited.

The elimination of the subroutines generally generates an linear increase in the size of the bytecode, but if we have nested subroutines we might have a exponential increase in the size of the code. This problem was studied in [1] by taking some statistics from some example programs and they conclude that:

- It is not common to find nested subroutines in the code, which mean that the size increase is usually linear.
- The examples studied show the size of the bytecode of the method where the subroutines where eliminated was increased substantially but the increase of size of a whole program after the subroutine elimination is not a great deal.

For all this reasons the optimization obtained by use of subroutines, may not be enough to compensate the difficulties introduced. Sun has already eliminated the subroutines from the bytecode language for its new

Java KVM virtual machine, which is specially designed to work with resource-constrained devices. They decided to do this presumably to simplify the on-device bytecode verification (for more information see [7]). The KVM virtual machine comes with a tool for eliminating the subroutines form the bytecode.

3.2 Eliminating the subroutines

Different strategies could be used to eliminate subroutines from the bytecode. A simple one could be performing inline substitution of the subroutine bytecode for each subroutine invocation (`jsr`). This would be the technique used by our transformation. Other options have been studied in [1]. This technique is simple but not trivial, there are some problems that must be considered:

- *Flow control instructions*: The inline substitution replace on instruction (`jsr`) with all the instructions of the subroutine. As a result, the bytecode after the `jsr` should be shifted, so all the addresses of the flow control instructions (like `goto` or `if_cond`) may need to be recalculated. Instructions like `lookuptable` that manage relative addresses also may need to be recalculated.
- *Exception Handlers*: For the same reasons, the addresses of the protected area and the location of the handler must be also recalculated after the transformation.
- *Exception Handlers inside the subroutine*: Each copy of the subroutine must have the same handlers as the original subroutine in order to preserve the semantics, so a copy of each exception handler must be done and the protected area must be recalculated to cover the instructions in the subroutine copy.
- *Exception Handlers for the finally block*: Each `finally` block has its own handler, which covers the `try` block and all of the `catch` blocks. This handler is a special handler that can catch any type of exception. If any unhandled exception is raised in the `try` or in any of the `catch` blocks, this handler will catch that exception and will execute the `finally` block before raising it again, see sections 7.12 and 7.13 of [2]. In the Figure 2.1 the protected area of this handler covers lines 0 to 16 and the handler starts at line 17.

  How the inline substitution should take place is a delicate matter, because if we follow our intuition and only insert the bytecode of the subroutine at the position where the subroutine call is, we would be changing the semantics of the `try-catch-finally` block. The problem is related to the exception handlers because all the handlers that are over the `try` block, including the `finally` handler are not supposed to catch exceptions produced inside the `finally` block bytecode, but if this inline substitution is preformed in place, then the bytecode of the subroutine would be inside the `try` block, and so inside the protected area of all the handlers that cover the `try` block (including the `finally` block itself).So if an exception is raised inside this inserted code, it might be cached by any of the handlers or even the `finally` block. This problem can be easily overcome, a solution will be presented in next section. This problem is well documented in [14].

The elimination of the subroutines from the methods' bytecode simplifies the work that must be performed by the bytecode verifiers. The verification algorithms would not have to deal with polymorphism introduced by the subroutines any more. After the subroutine elimination we would have one copy of the subroutine bytecode for each subroutine call, so each subroutine call would access its own copy of the subroutine. State merging is not needed any more.

4. The transformation

4.1 The platform

The subroutine elimination transformation was implemented and tested over a formal executable specification of the Java Card Virtual Machine that is implemented in the COQ Proof Assistant [10, 11]. This formal specification was built in the Certicartes project [8].

This formal specification allows the execution of any Java Card program (with some restrictions, see [8]). It treats the whole set of Java Card instructions and it considers every important aspect of the platform. The formalization may be used to reason about the Java Card platform itself and to prove properties about

Java Card programs. These properties are of great value for the proof of correctness of our transformation, they allow us to test and to reason about the transformation. For more information see, [8].

COQ is a proof assistant based on the Calculus of Inductive Constructions. It combines a specification language (featuring inductive and record types) and a higher-order predicate logic (via the Curry-Howard isomorphism). All functions in COQ are required to be terminating, recursive functions must be defined by structural recursion. For more information see, [10, 11].

4.2 Brief description of the JCVM

Java Card programs are Java programs with some constraints (because the limitation in resources of the platform) and some special features. These programs are compiled with a standard Java compiler and then must go through another transformation process that checks that the constraints are satisfied and also merge all the class files of the application package that is being loaded into a single file, called the CAP file. The CAP file is the loading unit in Java Card, it contain all classes and interfaces of a single package.

Certicartes has defined its own representation of the CAP file (see [8]) and it provides the JCVM Tools that converts a standard CAP file into a COQ expression that represents it (this expression is of the type `jcprogram`). This new CAP file then can be loaded and executed in COQ. When the JCVM Tool generates this new CAP file representation it also performs some key transformations:

- A complete resolution and elimination of the constant pool component. The Certicartes JCVM works with a structured representation of the CAP files and not directly with a byte stream format that forms the CAP files. This structure eliminates the problems of having branches to the middle of an instruction, which is a problem that usually has been taken into account to implement the subroutine elimination. The JCVM Tools perform a complete resolution and linking.
- Transformation of relative target addresses into absolute target addresses.

Here we only mention the transformations that are relevant for our subroutine elimination problem, for more information see [8].

The tool also ensures that some invariants required by Sun's specification hold. The two invariants relevant for us are:

- The target address of branching instructions remains within the calling method
- The apparition order of the handlers inspecting the same block of code is respected by the transformation, so if we have a `try` with two `catches`, the first `catch` will appear first in the list of handlers

4.3 Restrictions over the bytecode

Usually JVML (or JCVML) programs are written in Java and then translated to JVML bytecode by the Java compiler, but these programs may be written directly using the bytecode language as well. Programs written directly at bytecode level have much more freedom of what they do and how they do it. They might be built in a completely unstructured way. On the other hand, programs generated by the compiler usually have a well defined structure, as they are automatically translated from Java code.

Constructing and proving the correctness of a transformation that deals with all kind of bytecode is a complex task and demands a great amount of effort. Considering the fact that almost all the bytecode (if not all) will be generated by a compiler, it may be more convenient to put all the efforts in the compiled bytecode. Our transformation is focused on well structured bytecode (the general case would be presented as future work), so to define which is the bytecode accepted by our transformation we give a set of properties that it must be fulfill in order to be correctly transformed. The definition of these restrictions was based on the informal ideas presented in the Java specification [2] (of how a `try-finally` block should be compiled) and also these restrictions were checked against the bytecode generated by the Sun JDK 1.3.1 (for Linux), and all tests done show they were fulfilled.

Restrictions for the structure of the bytecode:

- Each subroutine call (`jsr` instruction) must be before the subroutine body in the method's bytecode.
- We define the body of a subroutine as follows: it starts with a `store` instruction (that removes the return address of the subroutine of the stack into a local variable) and ends with a `ret` instruction. All instructions after the `store` instruction and before the `ret` instruction are part of the body of the subroutine (including the `store` and `ret` instructions).
- Each jump made inside a subroutine must keep the execution inside the subroutine (including the `tableswitch` and `lookupswitch` instructions). Only with a `ret` instruction or an exception the execution can jump outside the subroutine.
- Each subroutine must have only one `ret` instruction (of its own, it can have more `ret` instructions in its body it there are nested subroutines) and it can't share it with another subroutine.
- The local variable used by the `store` instruction at the beginning of the subroutine must be only used to store its return address through the method's bytecode.
- Given two subroutines s1 and s2, one of the next three properties must be valid: the body and invocation of s1 are included in the body of s2, the body and invocations of s1 are completely outside the body of s2 or the body and invocations of s2 are included in the body of s1.
- The only way to stop executing bytecode inside the body of a subroutine is though the invocation of the subroutine with a `jsr` instruction starting at the first instruction of the body.

Restrictions for the handlers:

- For each handler, one of the next three properties must be true:
  - The protected area and the code of the handler are included inside the subroutine body
  - The subroutine body is completely inside the protected area of the handler.
  - The subroutine body and invocations are completely inside of the code of the handler

Most of these restrictions are meant to simplify the determination of the subroutine's body. Future work should eliminate these restrictions and consider bytecode where the body of the subroutine is not so well delimited. In [14] there is a well documented algorithm to determine the body of a subroutine.

4.4 Overview of the transformation

The goal of the transformation is eliminate all the subroutines form a given Java Card program, represented by a CAP file (`jcprogram` in Certicartes). To do this it must eliminate all occurrences of a subroutine from all the methods of all the classes in the CAP file. Formally, this transformation can be defined as function `T`.

```
T: jcprogram->jcprogram
```

Considering the fact that our final goal is proving the equivalence between a program `p` and `T(p)`, defining the transformation in this way may difficult the proof. A transformation of these characteristics where all subroutines in a program are eliminated implies changes all over the bytecode, increasing the proof's complexity. In order to reduce it, consider a new transformation `T'` that only eliminate the first occurrence (if there is any) of a subroutine in a given method. This transformation can be defined by:

```
T': jcprogram->cap_method_idx->jcprogram
```

`T'` receives a `jcprogram`, the index of one program's method (in Certicartes method's index are represented by `cap_method_idx`) and it return a `jcprogram`. The returned program identical to the original but the first occurrence of a subroutine was eliminated from the specified method. Each time the transformation is applied it eliminates one subroutine invocation. The transformation searches for the first `jsr` instruction in the bytecode and eliminate it. The elimination is preformed by inline substitution as it will be explained ahead. The subroutine called by this invocation would be eliminated when all its invocations were eliminated. In order to eliminate all the subroutines from a `jcprogram` the transformation must be applied a finite number of times. All possible changes introduced by this transformation would be only on the target method.

This transformation definition allows that the proof can be structured in a more convenient way. We can prove the equivalence when only one invocation is eliminated (when `T'` is applied) and then prove by induction in the number of subroutines invocations that when all invocations are eliminated the original and the transformed programs are equivalents. Proving the equivalence for transformation `T'` is simpler than for transformation `T` because it is well defined which parts of the program were affected by it.

The mechanism used to eliminate the subroutines is through inline substitution. As it was mentioned in Section 3.2, there are some considerations that must be taken when performing the substitution in order to preserve semantics of the `try-catch-finally` block. There are two different ways of doing the inline substitution. The first option is replacing the subroutine call by a copy of the subroutine bytecode. In order to preserve the semantics of the `try-catch-finally` block the handlers of the `finally` block must be in two in order to leave the copy of the subroutine outside the handler. This process must be repeated for each subroutine invocation. This approach have the disadvantage that it can produce an explosion in the number of handlers needed (the maximum number of handlers per method are 255). This problem is even worst if there are nested subroutines.

The second option is make a copy of the subroutine but instead of inserting it where the `jsr` instruction is, it would be inserted outside the protected area of the finally block's handler. By doing this there is no need to split the handlers. Also the `jsr` instruction must be replaced by a `goto` jumping to the address where the copy of the subroutine was inserted. In our transformation the copy is inserted at the end of the subroutine. Remember that we are eliminating one call to a subroutine at a time, so usually the copies of the subroutine will coexist with the subroutine itself.

Figure 4.1 shows an abstraction made over the structure of the method's bytecode. This abstraction is based on the restriction imposed over the bytecode in the previous section. The method's bytecode can be divided in four blocks.

Based on the restrictions imposed (in the previous section) over the bytecode, an abstraction in terms of blocks of instruction could be made. As shown in Figure 4.1, the bytecode of a method that include subroutines could be divided in four blocks. The first block goes from the first instruction up to the instruction immediately before the first `jsr` instruction founded in the bytecode. The second block goes from the instruction immediately after the first `jsr` up to the instruction immediately before beginning of the subroutine (before the `store`). The third block is from the instruction immediately after the first instruction of the subroutine (`store`) up to the instruction immediately before the last instruction of the subroutine (the `ret`). Note that this block represents the subroutine being eliminated. The forth block goes from the instruction immediately after the last instruction of the subroutine (the `ret`) up to the end of the method bytecode.

From now on we will adopt the next naming convention when referring to the components of the methods bytecode and some properties of it. The first invocation of a subroutine will be called first call (`fc`). The address of the subroutine invoked by the first `jsr` will be called body start (`bs`). The `bs` is the first instruction of the subroutine being eliminated and it is always a `store`. The address of the last instruction of the subroutine will be called body end (`be`) and it is always a `ret`. The number of invocations to the subroutine inside the method's bytecode will be called number of calls (`nc`).

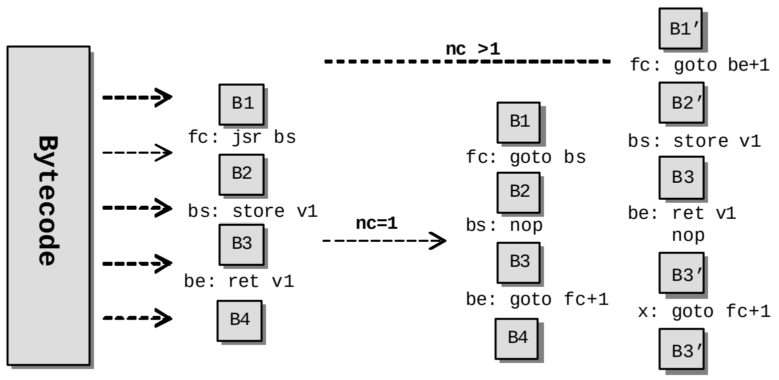


Figure 4.1: The transformation

Based on the abstractions presented in Figure 4.1, the transformation `T'` can be divided in three cases:

- There are no subroutine calls in the bytecode. In this case the method keeps unchanged.
- There are subroutine calls in the bytecode and there are only one call to the subroutine been eliminated, so `nc=1`. As shown in Figure 4.1, in this case the subroutine is eliminated only by replacing the `jsr` by a `goto` jumping to the same address as `jsr`, witch is `bs`. Also the `store` instruction is replaced by a `nop` and the `ret` instruction is replaced by a `goto` jumping to the instruction immediately after the `jsr`, witch is `fc+1`. The reason to replace the `store` by a `nop` is to simplify the proof as it will be explained in section 5.1.2. The rest of the code and the exception handlers are not affected by the transformation.
- There are subroutine calls in the bytecode and there are more than one call to the subroutine been eliminated, so `nc>1`. As shown in Figure 4.1, in this case the block B3 (the subroutine body without the first an last instructions, the `store` and `ret`) will be cloned and it will be inserted immediately after the end of the subroutine, after `be`. Between the `ret` at be and B3' there is a `nop`, witch replace the `store` instruction of the subroutine body as explained in the previous case. Also after the B3' there is a `goto` jumping to `fc+1` witch replace the `ret` (also as explained in the previous case). All the address of the branching instructions in blocks B1, B2, B4 and B3' should be corrected. The exception handlers inside B3' should be cloned. Other exception handlers should be recalculated.

The actual implementation of this transformation in COQ is a bit different. The transformation is splited in two, one function that transforms the bytecode and one function that transform the handlers. By doing it in that way it is easier to establish that after applying `T'` the only areas of the program affected are the bytecode and the handlers of the method received as a parameter. The definition of those functions is as follows:

```
Definition subroutElim : (list Instruction)->subroutine->(list Instruction)

Fixpoint handler_correction [s:subroutine; l:(list handler_type)]:
  (list handler_type)
```

Both functions receive as a parameter a record of the type `subroutine`. This record stores pre-calculated information about the bytecode of the method been processed. It contains the `fc`, `bs`, `be` address it also contains the `nc`, the number of the local variable used by the subroutine to hold the return address, the body of the subroutine been eliminated ready to perform the inline substitution (it already replace the `store` by the `nop`, the `ret` by the `goto` and perform the address correction of the branch instructions) and the length of the body. The COQ definition of the record is as follows:

```
Record subroutine : Set := {
    firstcall: bytecode_idx; (*Address of the first invocation*)
    numbercalls: nat; (*Number of calls to the subroutine*)
    retaddr: locvars_idx; (*Number of the local variable to hold the Return Address*)
    bodystart: bytecode_idx; (*Address of the subroutine begin*)
    bodyend: bytecode_idx; (*Address of the subroutine end*)
    body: (list Instruction); (* Bytecode of the sub-routine *)
    bodylen: nat; (* Length of the bytecode of the sub-routine *)
    processing: bool (*For internal use of the analyze algorithm*)
}.
```

The function that pre-calculates all this values is called `AyzStruct` (analyze structure) and it is defined as follows:

```
Definition AyzStruct : (list Instruction)->subroutine
```

The subroutine elimination function `subroutElim` receives a list of instructions (the bytecode) and it returns another list of instructions where the first invocation to a subroutine was eliminated. The `handler_correction` function receives a list of handlers and returns another list of handlers where all the recalculation or cloning needed has been preformed.

5. Reasoning on the correctness of the transformation

Working on top of COQ and having the full implementation of de JCVM makes possible to reason on the correctness of the transformation proposed. By correctness we mean that the transformation should preserve the semantics of the Java program being transformed and also the BCV acceptance. The BCV acceptance state that if a given program is accepted by the BCV before applying the transformation it must also be accepted after applying the transformation. Note that in the opposite direction the property do not necessary have to hold.

Section 5.1 presents the work done up to the moment in the formal proof of equivalence. Section 5.2 gives an intuitive idea of why the BCV will accept the transformed program.

5.1 Equivalence proof

5.1.1 Proof's goals

The goal is proving that the transformation preserves the semantics of a Java program. This will be done by proving for a program `p`, the execution of the program with subroutines and without subroutines (after the transformation) lead to equivalents states. Those states are not equal, the differences are in the frames of execution associated to the transformed method, but those differences don't affect the result of the execution of the methods. This would be explained in more detail in section 5.1.3.

5.1.2 Proof strategy

The strategy that we will use for the proof will be based on a commutation diagram. In order to prove the equivalence of both programs, we will prove that the execution of the original program commute with the execution of the transformed program. We take two special considerations to simplify the proof without losing generality in it. These considerations also contribute to a clearer and more direct proof.

The first consideration is related to the strategy used in the definition of the transformation. Intuitively it is easier to prove that the two programs are equivalent if they are "similar". Defining the transformation to eliminate only the first subroutine invocation founded in the bytecode as it was defined in section 4.4 simplifies considerably the proof because the proof can be focused in one method and one subroutine invocation rather than the whole program.

The second consideration is related to the number of execution steps needed by the original and transformed programs to reach equivalent states. This number may be a different number for each program. The proof's complexity would be lower if we have the same number of steps. During the inline substitution when the subroutine body is cloned, the `store` instruction at the beginning of the subroutine is useless (because we don't have to deal with the return address any more), so it can be eliminated. The consequence of this elimination would be that the transformed program would need one less step to execute the subroutine. Ergo, to avoid this unpleasant problem the transformation instead of eliminating the `store` instruction it will replace it by a `nop` instruction. This instruction doesn't affect the state and it corrects difference in the steps. Having the same number of execution steps for both bytecode makes possible the next simplification; we can prove that the execution is equivalent for one step of execution rather than an arbitrary number of steps.

In order to avoid having useless instructions in the code like the `nop` instruction, it should be easy to implement another transformation that eliminates all the `nop` instructions from the code and then proving the equivalence of both programs. But this will be left as future work.

So with these two simplifications in mind we can build the commutation diagram as we can see in Figure 5.1. Given a `jcprogram` and a `state` we reach the same state `returned_state`^{t} by executing one step of the original code and then applying the state transformation function (`rtransf`) or by first executing the subroutine elimination transformation (`subroutElim`) and state transformation function (`transf`) and then execute one step over the transformed code.

$$\begin{tikzcd}
\texttt{(jcprogram, jcvm\_state)} \arrow[r, "\texttt{exec\_1\_step}"] \arrow[d, "\texttt{subroutElim + transf}"] & \texttt{(jcprogram, returned\_state)} \arrow[d, "\texttt{rtransf}"] \\
\texttt{(jcprogram}^t\texttt{, jcvm\_state}^t\texttt{)} \arrow[r, "\texttt{exec\_1\_step}"] & \texttt{(jcprogram}^t\texttt{, returned\_state}^t\texttt{)}
\end{tikzcd}$$

Figure 5.1: Commutative diagram

A `jcvm_state` is basically formed of a heap, a static heap and a stack of frames. A frame is responsible of having a method's state of execution. As it is shown ahead, in each frame there are the local variables, the operand stack and the program counter for a specific method. For more information refer to [8].

```
Record frame : Set := {
       opstack:(list valu); (* operand stack *)
       locvars:(list valu); (* local variables *)
       method_loc: cap_method_idx; (* location of the method *)
       context_ref: Package; (* context information *)
       analyzed: bool; (*This is for the BCV*)
       p_count: bytecode_idx (* program counter *)
}.
```

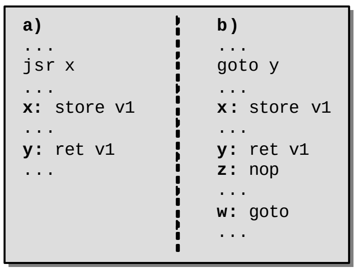
The function `exec_1_step` receives a program and a state and it execute the next step of execution based on the frame in the top of the stack. This function is defined as follows:

```
Definition exec_1_step: jcprogram->jcvm_state->returned_state :=
                              [cap:jcprogram; state:jcvm_state]
Cases (head (Snd (Snd state))) of
   (Some h) => (Cases (get_instruction (method_loc h) state cap) of
                       None => (AbortCode instruction_error state) |
                       (Some i) => (exec_instruction i state cap)
                  end) |
   (None) => (Normal state)
end.
```

5.1.3 State mapping functions

This section is meant to explain how the state mapping functions `transf` and `rtransf` actually work. The differences between a state corresponding to the original bytecode execution and the one corresponding to the transformed bytecode execution are located in the stack of frames. To be more precisely, they are located in the frames associated to the method being eliminated. Consequently to transform the original state (result of the execution of the original bytecode) to the transformed state (result of the execution of the transformed bytecode) we need to analyze each frame in the stack and transform it.

The components of the frame that may need to be transformed are the program counter, the set of local variables and the operand stack. To clarify why each of these components may need to be modified, let's consider the example shown in Figure 5.2.



a)	**b)**
...	...
jsr x	goto y
...	...
x: store v1	**x:** store v1
...	...
y: ret v1	**y:** ret v1
...	**z:** nop
	...
	w: goto
	...

Figure 5.2: Bytecode before (a) and after elimination (b)

In this example we can see the original bytecode (a) and the transformed bytecode (b) after the elimination of the first subroutine invocation. This example is in the hypothesis that there is more than one invocation to the subroutine being eliminated. Now let's see why the program counter may need to be transformed. If the original state's program counter is inside the subroutine (between `x` and `y`) then the transformed state's program counter would have two possible values. If the subroutine was called form the invocation at `fc` the program counter would be between `z` and `w`. In the other hand it the subroutine was called form other invocation then the program counter would be between `x` and `y`.

The state mapping transformation uses the value of the return address to decide which invocation calls the subroutine. The return address can be at the local variable or the operand stack, depending on the position in the bytecode. This is one of the reasons that motivate the restriction that each subroutine have its own local variable and that it can only be used by the `store` instruction at de beginning of the subroutine and the `ret` instruction at the end. Without this restriction we can't decide where the invocation was originated.

Similar considerations must be taken to transform the local variables. If we are executing inside the subroutine and it was called by the `jsr` instruction at `fc`, in the original state the local variable of the subroutine must be set with the return address. On the other hand, this local variable in the transformed state must be unset because the return address is not needed any more (in Certicartes an unset variable have a default value). If we are executing inside the subroutine but it was called form a different `jsr` than the one at `fc`, the value of the return address must be kept.

Local variables presents have another problem that is related to nested subroutines. The presence of nested subroutines inside the subroutine been eliminated demands that the return addresses stored by these subroutines in the local variables may need to be recalculated. These addresses are recalculated depending where the subroutine was called (this can be know examining the local variable at the original state). Consequently the transformation must scan the set of local variables and recalculate all return addresses that need so.

When the subroutine has just been called the operand stack stores the return address until the `store` instruction remove it. This return address may need to be removed form the original state; this depends on who made the call to the subroutine. If the called was made form `fc` the return address must be removed, in any other case it must be kept. The operand stack also has the same problem of the local variables with nested subroutines. This implies that the transformation must scan the operand stack and transform return addresses that need so.

In the case where there is only one invocation to the subroutine, the state transformation function is simpler. The program counter never needs to be transformed, because there are not significant changes in the structure of the bytecode (see Figure 4.1). We still have to check if we have to unset the return address from the local variable or remove it from the operand stack, but we don't have the nested subroutine problem any more.

Both functions `transf` and `rtransf` use the same state transformation algorithm, the only difference is that `transf` process a `jcvm_state` [8] and `rtransf` process a `returned_state` [8]. Actually the functions implemented to make these transformations were:

```
    s2s: cap_method_idx->subroutine->stack->stack                   (* impl. transf  *)
    r2r: cap_method_idx->subroutine->returned_state->returned_state (* impl. rtransf *)
```

The first one `s2s` (state to state) receive the index of the method being transformed, a subroutine element (it is a record with information about the method being transformed), and the stack of frames and return a transformed stack of frames. The second one `r2r` receive also the index of the method, a subroutine element and a `returned_state` and return a transformed `returned_state`. `r2r` uses `s2s` to perform the state transformation.

5.1.4 Testing the commutation experimentally

Now that all the functions of the commutation diagram (Figure 5.1) are defined, the property that must be satisfied to prove the commutation can be established. This property would be as follows:

```
    (r2r mid a (exec_1_step jp (sh, (hp,(cons f lf)))))=
                 (exec_1_step jpt (sh, (hp, (s2s mid a (cons f lf)))))
```

In the property `mid` represents the index of the method where the elimination would be preformed. The original `jcprogram` program is represented by `jp` and the transformed program by `jpt`. The state is represented by giving its components which are `hp` the heap, `sh` the static heap and `(cons f lf)` the stack of frames. Note that the stack of frames must have at least one frame represented by `f`.

The executablility of the Certicartes JCVM platform allow to experimentally test the commutation of the diagram using the functions previously defined `subroutElim`, `handler_correction`, `r2r`, `s2s`, `exec_1_step`. We implement a state equivalence check function that receives as input parameter the original program, the number of steps (`n`) to execute and the number of subroutine eliminations to perform (`m`). This function checks that for each execution step (between `0` and `n`) the original program commute with the transformed program. The transformed program is the result of applying `subroutElim` `m` times to the method).

These tests help us to detect some bugs in our functions in an early stage, avoiding the unpleasant moment of finding them during the proofs, which may result in having to make the whole proof again.

5.1.5 The theorem of equivalence

The theorem of equivalence ensures that the commutation property presented in the previous section holds for an arbitrary combination of a program and a state. The statement of the theorem is as follows:

```
1  Theorem EQ:
2  (jp:jcprogram)(sh:sheap)(hp:heap)
3  (lf:(list frame))(f:frame)(m, mti: Method)

4  let state = (sh, (hp, (cons f lf))) in
5  let mid = (method_loc f) in
6  let a = (AyzStruct (bytecode m)) in
7  let ltr = (subroutElim (bytecode m) a) in
8  let h = (handler_list m) in
9  let ht = (handler_correction a (handler_list m)) in
10 let mt = (mod_hb m ht ltr) in
11 let jpt = (Build_jcprogram
               (classes jp)
               (l_update_nth (methods jp) mid mt)
               (interfaces jp)) in
12 ((Some Method m)=(Nth_elt (methods jp) mid))->
13 ((Some Method mti) = (Nth_elt (methods jpt) mid))->
14 (mti=mt)->

15 (checkStructure (bytecode m))->
16 (handlerSub h (bodystart a) (bodyend a))->

17 (r2r mid a (exec_1_step jp state))=
     (exec_1_step jpt (sh,(hp,(s2s mid a (cons f lf))))).
```

The arbitrary program and state are represented by the `jp` and `state` variables respectively (line 4). The state is specified by its components, the static heap (`sh`), the heap (`hp`) and stack of frames. Note that the stack of frames must have at least one frame (`f`). The rest of the stack is represented by `lf` which is a list of frames.

The subroutine elimination is preformed over the method `m` of `jp`. The method `m` is at position `mid`, at line 12. The result is a new program `jpt` where only the bytecode and handlers of the method `m` were modified. The transformation is applied to the bytecode (line 7) and the handlers (line 9) of the method `m` separately, giving as a result a new bytecode (`ltr`) and a new list of handlers (`ht`). The transformed method `mt` is the result of replacing `ltr` and `ht` with `m's` bytecode and `m's` list of handlers respectively. In line 11 the `jpt` program is constructed by replacing original method `m` by the transformed method `mt`.

An extra restriction is imposed in line 5; the frame in the top of the stack (`f`) must be associated to the method `m`. Consequently this theorem will prove the commutation of the diagram when the next step of execution is over method `m`. The general statement shouldn't impose this restriction, it should cover this case and the case where the next step of execution is over a frame not associated to `m`. Definitely, the case covered by the theorem is most challenging of both. The only interesting part of the other case is when the next instruction to execute raises an exception and it is handled by the transformed method. The other cases can be trivially proved because those methods were not affected by the transformation. In our theorem statement we opt for leaving this case out because we consider that it didn't really contribute to the proof. This case will be proposed as future work.

The theorem also imposes three different groups of restrictions:

- Restrictions over the structure of the bytecode of the transformed method. Those restrictions are imposed by the `checkStructure` term at line 15.
- Restrictions over the list of handlers of the transformed method. Those restrictions are imposed by the `handlerSub` term at line 16.
- Restrictions over the state where the commutation property should be valid. In order to simplify the theorem's statement these restrictions were not included in it.

Those restrictions are explained in greater detail above.

Restrictions over the bytecode

The aim of these restrictions is establish the structure that the bytecode of the method to be transformed must have. These restrictions were presented informally in the section 4.3. Formally they are defined by inductive predicates. The `checkStructure` is an inductive predicate that receives the bytecode of a method as a parameter and it checks that all the restrictions are satisfied by it. This predicate is defined as follows:

```
Inductive checkStructure: (list Instruction)->Prop
```

To exemplify how the restrictions are formally defined using inductive predicates, consider the first restriction imposed over the bytecode in section 4.3:

"*Each subroutine call (jsr instruction) must be before the subroutine body in the method's bytecode*"

The inductive predicate that defines this restriction is:

```
Inductive checkJSRltSTOREi : Instruction->nat->Prop :=
   cons_jsr_other: (i:Instruction)(n:nat) (isOther i) -> (checkJSRltSTOREi i n) |
   cons_jsr_jsr: (b:bytecode_idx)(n:nat) (gt b n)->(checkJSRltSTOREi (jsr b) n) |
   cons_jsr_store: (t:type)(l:locvars_idx)(n:nat) (checkJSRltSTOREi (store t l) n) |
   cons_jsr_load: (t:type)(l:locvars_idx)(n:nat) (checkJSRltSTOREi (load t l) n) |
   cons_jsr_ret : (l:locvars_idx)(n:nat) (checkJSRltSTOREi (ret l) n).

Inductive checkJSRltSTORE: (list Instruction)->Prop :=
   nil_chkJltS: (checkJSRltSTORE (nil Instruction)) |
   cons_chkJltS: (i:Instruction)(li,lr:(list Instruction))(n:nat)
                 (checkJSRltSTORE li)->(p_length li n)->(p_ins_back i li lr)->
                 (checkJSRltSTOREi i n)->(checkJSRltSTORE lr).
```

This predicate checks that each `jsr` instruction points to an address greater than its own address. If this predicate is fulfilled for all `jsr` instruction the body of the subroutines will be always after its calls.

The inductive predicate is divided in two different predicates. `checkJSRltSTOREi` receives a instruction and its position in the bytecode, and it checks if the restriction is fulfilled. `checkJSRltSTORE` receives a list of instructions and i checks that the restriction holds for all instruction. `isOther`, `p_length`, `p_ins_back` are all auxiliary inductive predicates.

Restrictions over the handlers

These restrictions check if the given list of handlers fulfills the handler's restrictions presented informally in the section 4.3. These restrictions are also formally defined by an inductive predicate, called `handlerSub`. It receives as a parameter a list of handlers, the address where the subroutine start (`bs`) and the address where the subroutine ends (`be`); and it checks that all the handlers in the list are inside or outside the subroutine.

In Certicartes a handler is defined by the structure `handler_type`. It is defined as follows:

```
Definition handler_type := (bytecode_idx*bytecode_idx*class_idx*bytecode_idx).
```

The first two elements define the range where the exception handler is active, the protected area. The third element defines the class of exceptions that the handler must catch, whereas the last element points to the first instruction of the handler code that should be executed when the exception is catched. For more information see [8].

The predicate `handlerSub` is defined as follows:

```
    Inductive handlerSub: (list handler_type)->bytecode_idx->bytecode_idx->Prop :=
      nil_hSub: (b, e:bytecode_idx) (handlerSub (nil handler_type) b e) |
      cons_hSub1: (l:(list handler_type))(h:handler_type)(b, e:bytecode_idx)
                  (handlerSub l b e)->
                  ((lt (Fst h) b)/\(lt (Fst (Snd h)) b))->
                  (lt (Snd (Snd (Snd h))) b)\/(gt (Snd (Snd (Snd h))) e)->
                  (handlerSub (cons h l) b e) |
      cons_hSub2: (l:(list handler_type))(h:handler_type)(b, e:bytecode_idx)
                  (handlerSub l b e)->
                  ((gt (Fst h) e)/\(gt (Fst (Snd h)) e))->
                  (gt (Snd (Snd (Snd h))) e)->
                  (handlerSub (cons h l) b e) |
      cons_hSub3: (l:(list handler_type))(h:handler_type)(b, e:bytecode_idx)
                  (handlerSub l b e)->
                  ((lt (Fst h) b)/\(le e (Fst (Snd h))))->
                  (gt (Snd (Snd (Snd h))) e)->
                  (handlerSub (cons h l) b e) |
      cons_hSub4: (l:(list handler_type))(h:handler_type)(b, e:bytecode_idx)
                  (handlerSub l b e)->
                  ((gt (Fst h) b)\/(eq bytecode_idx (Fst h) b))/\(le (Fst (Snd h)) e)->
                  (gt (Snd (Snd (Snd h))) b)/\(lt (Snd (Snd (Snd h))) e)->
                  (handlerSub (cons h l) b e).
```

The predicate have four possible constructors:

- `conshSub1`: In the case that both the protected area and the handler code are before bs or the protected area is before bs and the handler code is after be.
- `conshSub2`: In the case that both the protected area and the handler code are after be.
- `conshSub3`: In the case that the subroutine is inside the protected area of the handler and the handler code is after be.
- `conshSub4`: In the case that both the protected area and the handler code are inside the subroutine.

Restrictions over the state

The assertion that the commutation diagram should hold for all state is not completely true, because not all possible states are valid JCVM states. By valid JCVM states we mean states that can be reached by a valid `jcprogram` that fulfill the restrictions established in section 4.3. The restriction defined in this sub-section defines the set of valid states in which the commutation diagram will hold. In order to simplify the equivalence theorem's statement this restrictions were not included in it.

Some examples of the restrictions that where imposed over the states are:

- In every frame associated to the method where the transformation was applied, the local variable used by the each subroutine to store the return address must exist. The statement of this restrictions is as follows:

  ```
  ((fr:frame) (method_loc fr) = mid ->
              {v:valu | (Nth_elt (locvars fr) (retaddr a)) = (Some valu v)})
  ```

- The next restriction is only valid in the case where number of calls is one. It establishes that for every frame associated to the method where the transformation was applied, subroutine instruction (the `store`) is about to be executed, a return address must be on the top of the stack.

This return address must be pointing to the address fc + 1. The statement on this restriction is as follows:

```
((fr:frame) (method_loc fr) = mid ->
            (p_count fr) = (bodystart a) ->
            (EX lv:(list valu) |
             (opstack fr) =
              (cons ((Prim ReturnAddress),(inject_nat (S (firstcall a)))) lv)))
```

- The local variable used by instructions like store must exist in every frame that is associated the method where the transformation was applied. The statement of this restriction is as follows:

  ```
  ((fr:frame) (method_loc fr) = mid ->
            {v:valu | (Nth_elt (locvars fr) l) = (Some valu v)})
  ```

  As it was explained at the beginning of this section, the theorem will only prove for one step of execution over a frame associated to the method affected by the transformation. This is the reason why this restriction only consider the frames affected by the transformation.

Another restriction imposed over the state says that when the program counter is at bs (so the subroutine has just been invoked) the value of the local variable that keeps the return address must be unset. To be more precisely the value will be the default value of the local variable which is represented by `default_valu`. This restriction means that the subroutine only can be called one time per method execution. Clearly this is a limitation for a generic bytecode, but it is not for the bytecode generated to compile the `finally` block. When the bytecode is from a `finally` block a subroutine is usually called one time per method execution. The reason to need this restriction is that if at bs the local variable could take different values, the state mapping function will not have enough information to decide which is the correct value. To know the value it would need to simulate the execution of the method or use another complex mechanism. This statement of this restriction is as follows:

```
((fr:frame)(v:valu) (method_loc fr) = mid ->
                    (p_count fr) = (bodystart a) ->
                    (Nth_elt (locvars fr) (retaddr a)) = (Some valu v) ->
                     v = default_valu)
```

Analyzing a different approach for avoid this restriction will be leaved as future work.

Only a portion of all the restrictions have been presented here. Most of the state restrictions appear while the proof was developed. Since the proof has not been developer completely, it is highly probable that many state restrictions will appear with the rest of it.

5.1.6 Proof structure

Partitioning

The definition of the structure of the proof was mainly directed by one goal, reduce and manage the proof's complexity. To address this issue the proof is structured in five layers (see Figure 5.3). Each layer proves a specific sort of property. To prove a property in layer n we will use the properties of the lower layers.

The first four layers are also partitioned vertically. These divisions are based on the value of the field `nc` of the `subroutine` record. The division is based on the three cases presented in Section 4.4, which are `nc=0` (no subroutine calls), `nc=1` (only one subroutine call) and `nc>1` (more than one subroutine calls).

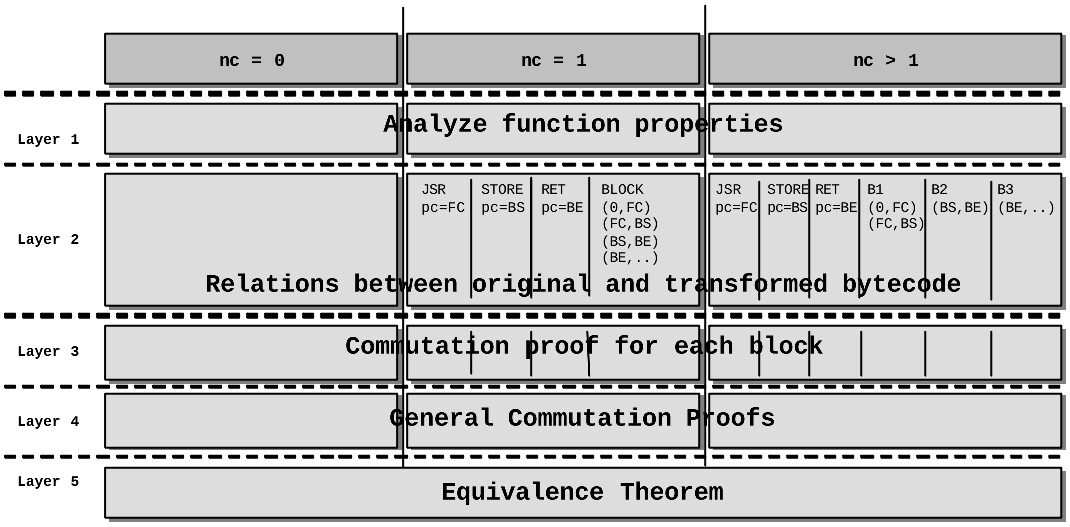

nc = 0
nc = 1
nc > 1
Layer 1
Analyze function properties
Layer 2
JSR pc=FC
STORE pc=BS
RET pc=BE
BLOCK (0,FC) (FC,BS) (BS,BE) (BE,..)
JSR pc=FC
STORE pc=BS
RET pc=BE
B1 (0,FC) (FC,BS)
B2 (BS,BE)
B3 (BE,..)
Relations between original and transformed bytecode
Layer 3
Commutation proof for each block
Layer 4
General Commutation Proofs
Layer 5
Equivalence Theorem

Figure 5.3: Proof's structure

Each layer has a specific objective, those objectives are:

- *Layer 1*: It establishes the relation between the bytecode of the method to be transformed and the `subroutine` record obtained by applying the `AyzStruct` function to it. The main goal if this layer is to precisely establish the mining of the analytical information generated by the `AyzStruct` function. This layer is composed by ten lemmas that state the properties. Those lemmas are proved based on the `AyzStruct` function. By working on top of this layer, we don't have to worry any more about the `AyzStruct` function in the proofs of the subsequent layers

- *Layer 2*: It establishes relations between the original and the transformed method. It basically states the changes in the bytecode and handlers introduced by the subroutine elimination transformation. This layer grasps the effects of the transformation, so by working on top of this layer we don't have to worry any more about the transformation's particular implementation.

  The effects of the transformation are particular for each case `nc=0`, `nc=1` and `nc>1`. The case `nc=0` where nothing change is trivial, both bytecode are the same. The effects for the other two cases can be extracted from Figure 4.1. A summary of those effects is presented above.

  Case `nc = 1`:

  - Blocks B1, B2, B3 and B4 remains unchanged
  - At the position of the `jsr` instruction (`fc`) there is a `goto` instruction with the same target address of the `jsr`
  - At the position of the `store` instruction (`bs`) there is a `nop` instruction
  - At the position of the `ret` instruction (`be`) there is a `goto` instruction. Its target address is `fc +1`

  Case `nc > 1`:

  - Blocks B1' and B2' basically are the same as B1 and B2 respectively but the address of the brunch instructions are recalculated
  - At the position of the `jsr` instruction (`fc`) there is a `goto` instruction. Its target address is `be + 1`
  - At the position of the `store` instruction (`bs`) there is the same `store` instruction
  - Block B3 of the transformed bytecode remains unchanged
  - At the position of the `ret` instruction (`be`) there is the same `ret` instruction
  - At the position `be + 1` there is a `nop` instruction
  - Block B3' basically is the same as B3 but the address of the brunch are shifted
  - At the position `be` + body length (is the length of the subroutine body including the `store` and `ret` instructions) there is a `goto` instruction. Its target address is `fc +1`

- Block B4’ basically is the same as B4 but the address of the brunch instructions are recalculated

- *Layer 3*: It establishes correspondences among the original and transformed bytecode. To understand what correspondence mean here, let’s analyze the state mapping function. As it was previously said, one of the elements of a frame that the state mapping function may transform is the program counter. This new program counter would point to the correspondent instruction of the transformed bytecode. That is precisely what correspondence means. An instruction in the original bytecode may have one or two correspondent instructions in the transformed bytecode. This layer states the correspondences and also proves that the execution of each pair of correspondents commutes.

  Following the relations established by the previous layer we state the correspondences between blocks of instructions or between instructions individually. The correspondences are shown in Figure 5.4.

Original	Transformed (nc=1)	Transformed (nc>1)
B1	B1	B1’
jsr	goto	goto
B2	B2	B2’
store	nop	store, nop
B3	B3	B3, B3’
ret	goto	ret, goto
B4	B4	B4’

Figure 5.4: Bytecode correspondences

  This figure state that block B1’ in the case nc > 1 corresponds to block B1 in the original bytecode. If the program counter is inside B1 the state mapping function transforms it to point to its correspondent instruction in B1’. In the other hand the block B3 of the original bytecode has two correspondent blocks when nc > 1.

  In order to structure the proofs in layers 2 and 3 we introduce an extra vertical subdivision, in addition to the original one (see Figure 5.3). This subdivision only affect the cases nc = 1 and nc > 1. These divisions are related to the correspondences of Figure 5.4. Each division has a lemma that proves its commutation. In the case of nc = 1 we have four lemmas or subdivisions, one for each instruction (`jsr`, `store`, `ret`) and one for the four blocks. In the case of nc > 1 we have six subdivisions, again one for each instruction and then one for the first two blocks, other for the subroutine body (B3) and another for the last block (B4).

- *Layer 4*: For each case nc = 0, nc = 1 and nc > 1, it proves that the execution of the original method commute with the execution of the transformed. This proof is done by performing induction on the program counter value.

  Here again the state mapping function has a crucial role, because for each program counter value the function would transform it to the correspondent in the transformed bytecode. This means that for each instruction in the original bytecode the state mapping function give us the correspondent instruction of the transformed bytecode. The commutation of these two instructions is trivially proved with the lemmas of the previous layer.

- *Layer 5*: It proves the Equivalence Theorem based on previous layers.

Quoting the proof

It is a fact that proving all the lemmas and theorems for the five layers is a huge task, so we decide to quote the proof’s scope. The proof will be done for a subset of the JCVM instruction set. The instructions chosen to be in this subset are the most affected by the transformation. These instructions are:

- `jsr`, `ret` and `store`: Needed to consider nested subroutines
- `push` and `pop`: Because they manipulate the operand stack, which is one of the instructions affected by the state mapping function
- `store` and `load`: Because they manipulate the local variables, which is the other instructions affected by the state mapping function
- `goto`: As a brunch instruction it is directly affected by the transformation's address recalculation
- `invokeinterface`: Because it is a method call function that can potentially raise exceptions (when the reference of the invoked object is null) or create a new frame in the frame stack (the normal execution scenario)
- `nop`: The no operation was only included because is used by the transformation

The `tableswitch` or `lookupswitch` instructions are also particularly affected by the transformation but we decide not to include them at this time, we propose this as future work.

The final instruction subset is:

```
Instruction_subset =
       {jsr, ret, pop, push, store, load, goto, nop, invokeinterface}
```

Examples

Now we will exemplify the first four proof layers by giving the statement of one or more it's lemmas. The theorem of the layer number five is the general theorem already introduced. For the first layer we present two lemmas.

```
Lemma eqAyz1: (m:Method)

              let a = (AyzStruct (bytecode m)) in
              (lt (S (firstcall a)) (bodystart a))/\(lt (bodystart a) (bodyend a))/\
              (~(bodystart a)=(0))/\(~(bodyend a)=(0))->
              (gt (numbercalls a) (0))->
              (checkStructure (bytecode m))->

              {idx: bytecode_idx |
               (Nth_elt (bytecode m) (firstcall a)) =
                                   (Some Instruction (jsr idx))/\(idx=(bodystart a))}.
```

This lemma states that the instruction whose position is determined by the field `fc` of the structure returned by `AyzStruct` must be a `jsr` instruction. This is valid only if the method's bytecode have the correct structure and at least one subroutine was found.

```
Lemma eqAyz7: (m:Method)(i:nat)(ins:Instruction)

              let a = (AyzStruct (bytecode m)) in
              (lt (S (firstcall a)) (bodystart a))/\(lt (bodystart a) (bodyend a))/\
              (~(bodystart a)=(0))/\(~(bodyend a)=(0))->
              (gt (numbercalls a) (1))->
              (checkStructure (bytecode m))->
              (gt i (1))/\(lt i (pred (bodylen a)))->
              (Nth_elt (bytecode m) (plus (bodystart a) i)) = (Some Instruction ins)->

              (Nth_elt (body a) i) = (Some Instruction (address_recalc ins a)).
```

This lemma states that in the case where `nc>1` the instructions in the field named body of the structure returned by `AyzStruct` are the subroutine's body instructions and that the address of those instructions have been recalculated.

For the second layer we give other two examples:

```
Lemma ncOne1: (m:Method)(i:nat)

              let a = (AyzStruct (bytecode m)) in
              let ltr = (subroutElim (bytecode m) a) in
              let mbc = (bytecode m) in
              (checkStructure (bytecode m))->
              (lt (S (firstcall a)) (bodystart a))/\(lt (bodystart a) (bodyend a))/\
              (~(bodystart a)=(0))/\(~(bodyend a)=(0))->
              ((numbercalls a) = (1))->
              (~i=(firstcall a))->(~i=(bodystart a))->(~i=(bodyend a))->
              (lt i (length (bytecode m)))->

              ((Nth_elt (bytecode m) i) = (Nth_elt ltr i)).
```

This lemma state that when we only have one subroutine call, the blocks B1, B2, B3 and B4 (Figure 4.1) of the original bytecode are not modified by the transformation.

```
Lemma ncOneJSR2: (m:Method;x:Instruction)(idx:bytecode_idx)

      let a =(AyzStruct (bytecode m)) in
      let ltr =(subroutElim (bytecode m) a) in
      (checkStructure (bytecode m))->
      (lt (S (firstcall a)) (bodystart a))/\(lt (bodystart a) (bodyend a))
      /\~(bodystart a)=(0)/\~(bodyend a)=(0)->
      (numbercalls a) = (1)->
      (Nth_elt ltr (firstcall a)) = (Some Instruction x)->
      (Nth_elt (bytecode m) (firstcall a)) = (Some Instruction (jsr idx))->
      x = (goto idx).
```

This lemma state that the `jsr` instruction located at `fc` will be replaced by a `goto` to the same target position.

For the third layer we give another two examples:

```
Theorem NC1_JSR: (jp:jcprogram)(sh:sheap)(hp:heap)
                 (lf:(list frame))(f:frame)(m, mti: Method)
    let state=(sh, (hp, (cons f lf))) in
    let mid = (method_loc f) in
    let a = (AyzStruct (bytecode m)) in
    let ltr = (subroutElim (bytecode m) a) in
    let h = (handler_list m) in
    let ht = (handler_correction a (handler_list m)) in
    let mt = (mod_hb m ht ltr) in
    let jpt = (Build_jcprogram
                 (classes jp)
                 (l_update_nth (methods jp) mid mt)
                 (interfaces jp)) in
    ((Some Method m)=(Nth_elt (methods jp) mid))->
    ((Some Method mti) = (Nth_elt (methods jpt) mid))->
    (mti=mt)->
    (checkStructure (bytecode m))->
    (lt (S (firstcall a)) (bodystart a))/\(lt (bodystart a) (bodyend a))/\
    (~(bodystart a)=(0))/\(~(bodyend a)=(0))->
    ((numbercalls a) = (1))->
    ((p_count f)=(firstcall a))->
    (r2r mid a (exec_1_step jp state)) =
                     (exec_1_step jpt (sh, (hp, (s2s mid a (cons f lf))))).
```

This theorem state that when we only have one subroutine call and the program counter of the actual frame of execution is at `fc`, the execution of the `jsr` instruction of the original code (in a given state) commute with the execution of the `goto` instruction of the transformed bytecode (in the transformed state).

```
Theorem NC1_EQB:  (jp:jcprogram)(sh:sheap)(hp:heap)
                  (lf:(list frame))(f:frame)(m, mti: Method)
    let state=(sh, (hp, (cons f lf))) in
    let mid = (method_loc f) in
    let a = (AyzStruct (bytecode m)) in
    let ltr = (subroutElim (bytecode m) a) in
    let h = (handler_list m) in
    let ht = (handler_correction a (handler_list m)) in
    let mt = (mod_hb m ht ltr) in
    let jpt = (Build_jcprogram
                  (classes jp)
                  (l_update_nth (methods jp) mid mt)
                  (interfaces jp)) in
    ((Some Method m)=(Nth_elt (methods jp) mid))->
    ((Some Method mti) = (Nth_elt (methods jpt) mid))->
    (mti=mt)->
    (lt (S (firstcall a)) (bodystart a))/\(lt (bodystart a) (bodyend a))/\
    (~(bodystart a)=(0))/\(~(bodyend a)=(0))->
    ((numbercalls a) = (1))->
    (checkStructure (bytecode m))->
    (~(p_count f)=(firstcall a))->(~(p_count f)=(bodystart a))->
    (~(p_count f)=(bodyend a))->
    (handlerSub h (bodystart a) (bodyend a))->
    (r2r mid a (exec_1_step jp state)) =
               (exec_1_step jpt (sh, (hp, (s2s mid a (cons f lf))))).
```

This theorem state that when we only have one subroutine and the program counter of the actual frame of execution is on a location inside B1, B2, B3 or B4 the execution of the instruction of the original bytecode at that position (in a given state) commute with the correspondent instruction in the transformed bytecode (in the transformed state),

For the forth layer we give one example, for the case where nc = 1.

```
Theorem NC1:  (jp:jcprogram)(sh:sheap)(hp:heap)
              (lf:(list frame))(f:frame)(m, mti: Method)
    let state=(sh, (hp, (cons f lf))) in
    let mid = (method_loc f) in
    let a = (AyzStruct (bytecode m)) in
    let ltr = (subroutElim (bytecode m) a) in
    let h = (handler_list m) in
    let ht = (handler_correction a (handler_list m)) in
    let mt = (mod_hb m ht ltr) in
    let jpt = (Build_jcprogram
                (classes jp)
                (l_update_nth (methods jp) mid mt)
                (interfaces jp)) in
    ((Some Method m)=(Nth_elt (methods jp) mid))->
    ((Some Method mti) = (Nth_elt (methods jpt) mid))->
    (mti=mt)->
    (lt (S (firstcall a)) (bodystart a))/\(lt (bodystart a) (bodyend a))/\
    (~(bodystart a)=(0))/\(~(bodyend a)=(0))->
    ((numbercalls a) = (1))->
    (checkStructure (bytecode m))->
    (handlerSub h (bodystart a) (bodyend a))->
    (r2r mid a (exec_1_step jp state)) =
               (exec_1_step jpt (sh, (hp, (s2s mid a (cons f lf))))).
```

This theorem prove that when nc = 1 the execution of the original program commute with the execution of the transformed program. This theorem is proved by induction on the program counter value and using the lemmas of the previous layer.

5.1.7 Proof status

The proof of this subset has not been finished yet; it is still a work in progress. The status of the proofs up to the moment is summarized in Figure 5.5. The dark grey areas represent the lemmas and theorems already proved. The light grey areas are the proofs that remain to be proved.

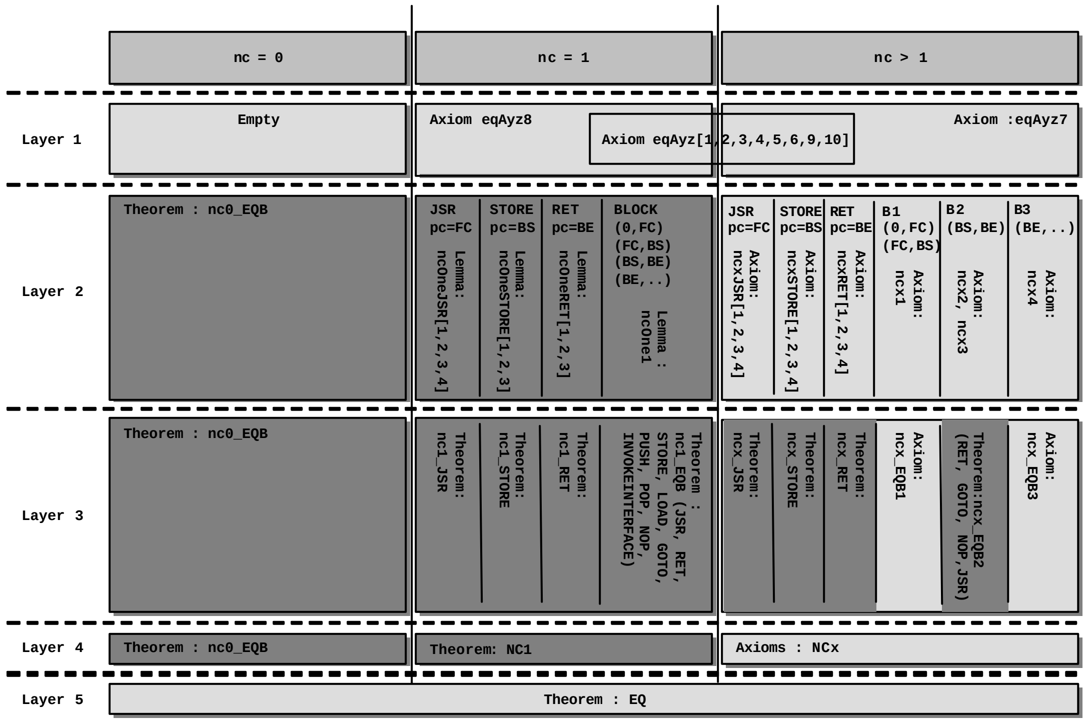


Figure 5.5: Proof status

The first layer is composed of ten lemmas that state properties about the analyze functions were replaced by axioms, because they don't contribute to the equivalence proof and its proof is very tedious. For the second layer (the one that state the relation between the original and transformed bytecode) was completely proved for `nc=0` and `nc=1`. In the case of `nc>1` the lemmas are still axioms that remains to be proved.

The proofs of commutation of the third layer were partially done. The theorems for the cases `nc=0` and `nc=1` were completely proved for the defined instruction subset. In the other hand for the case `nc>1` not all the theorems were proved. The main theorems proved for this case were:

- *ncx_JSR*: Prove commutation when the program counter is first call. As it was established in the table of Figure 5.4, this implies proving that the `jsr` instruction on the original bytecode is equivalent to the `goto` instruction of the transformed bytecode.
- *ncx_STORE*: Prove commutation when the program counter is body start. This implies proving that the `store` instruction on the original bytecode is equivalent to the `store` instruction and also to the `nop` instruction of the transformed bytecode.
- *ncx_RET*: Prove commutation when the program counter is body end. This implies proving that the `ret` instruction on the original bytecode is equivalent to the `ret` instruction and also to the `goto` instruction of the transformed bytecode.
- *ncx_EQB2*: Prove commutation when the program counter is inside the subroutine body. This implies proving that for each instruction of the subroutine body in the original bytecode commute with the correspondent instruction in B3 and B3' of the transformed bytecode. This proof was done only for the `ret`, `goto`, `jsr`, `nop` instructions. It should be completed to the rest of instructions of the defined subset.

The rest of the third layer remains to be proved.

The general commutation proofs in the forth layer were proved for the cases `nc=0` and `nc=1`, `nc>1` remains to be proved. The general theorem also remains to be proved. The theorems of both layers are easily proved having all the theorems in the previous layers.

To give a rough idea of the size of the proof in its actual status, let's see the number of lines needed to prove the first four layers:

- Layer 1: Axiom statements. 89 lines
- Layer 2: Bytecode relationships. 2080 lines
- Layer 3: Commutation proof. 5795 lines
- Layer 4: General commutation proof. 159 lines

- Bytecode's well formed restrictions (inductive properties). 412 lines
- Utilities lemmas and axioms. 895 lines

5.2 Byte Code Verifier acceptance

The main goal of this section is show that if a given program is accepted by the BCV before applying the subroutine elimination transformation it will be also accepted after applying it. The intention of this section is not giving a formal proof for this property; it only gives an intuitive idea why the new bytecode will is accepted by the BCV. Our analysis is based on the Sun BCV algorithm proposed at the JVM specification [2].

During the verification process most BCV algorithms have to approximate the state of the subroutine's first instruction by merging the states of all possible execution paths that reach it. By eliminating the subroutines BCV algorithms don't need to perform this approximation any more. Consequently the BCV algorithm enhances its precision and reduces the complexity and resources consumed by the verification process. At each iteration the transformation reduces the number of execution paths that reach a given subroutine.

Let's base divide the analysis of the transformation it in two cases, when there is only one call to the subroutine (`nc=1`) and when there is more than one (`nc>1`).

Case nc `= 1`

Before the transformation we have that:

- There is only one subroutine call which is the only way to access to the subroutine. This means that there is only one possible branch of execution that goes through the first instruction of the subroutine, which also implies that no approximation or state merge will happened at this point.

After the transformation we have that:

- The `jsr` is replaced by a `goto` pointing to the subroutine, so there is only one branch of execution that accesses the subroutine. The `goto` jumps to an instruction inside the bytecode and it can't have type checking problems.
- The `store` is replaced by a `nop`. The `nop` can't have type checking problems. The return address that the `jsr` push into the stack is not pushed any more by the `goto`. And the local variable used to store that address by the `store` is not used any more neither. There is a restriction that establishes that this local variable can't be used by any other instruction in the bytecode, so no type checking problems will result for that reason.
- The `ret` instruction is replaced by a `goto` pointing the `fc+1`. This `ret` instruction is the only way to return form the subroutine. The address `fc+1` is inside the bytecode.

To summarize, the number of branches that reach a given instruction remains the same and the replaced instructions don't introduce type checking problems.

Case nc `> 1`

Before the transformation we have that:

- There are at least two subroutines calls to the subroutine being eliminated.
- These calls represent the only execution branches that may access the subroutine.

- The BCV will merge the state of those branches at the first instruction of the subroutine. This may result on a loss of precision.

After the transformation we have that:

- The subroutine is cloned
- The `jsr` is replaced by a `goto`.
  - There is one less call to the original subroutine. The approximation may have the same precision or may improve it, but never have a loss of precision.
  - The `goto` points to the cloned subroutine and is the only way to access it. This implies that if that bytecode was previously accepted by the BCV now it must be accepted also. No approximation is needed to exe cute the same bytecode.
  - The `goto` can't produce type checking problems
- In the cloned subroutine the `store` is replaced by a `nop` instruction.
  - There is no need to manage the return address by the cloned subroutine, so the local variable previously used will be uninitialized. This is not a problem because the restriction that no other instruction could use this variable.
  - The `nop` instruction can't produce type checking errors.
  - The address correction assures that all jumps inside the method's bytecode keeps inside it.
- In the cloned subroutine the `ret` instruction is replaced by a `goto` instruction.
  - The address of the `goto` is `fc + 1` which is inside the bytecode
  - The `goto` instruction can't produce type checking errors.

The transformation doesn't introduce any buffer overflow or underflow problem. Before the transformation the stack was used to temporary store the return address only during the transition of executing the `jsr` instruction to executing the `store` instruction. This is not needed any more after the transformation.

6. Conclusions

Transformations like the one we build that transform programs developed to execute in a highly secure environment like Java Card are not allowed to have bugs. So a formal proof of the correctness of the transformation is not an option. Despite the fact that the proof is not completed yet, in the current status of it we acquire great confidence on the correctness of the transformation.

Working on top of a formal environment like COQ came with great benefits. In our case proving the semantic equivalence of the subroutine transformation helps us to discover bugs in the transformation. This was the case of the problem "Exception Handlers for the finally block" presented at section 3.2. In our first version of the transformation we didn't know about the existence that this problem, so that version of the transformation actually was changing the semantics of the Java program. During the proofs we realize about this problem and then we also find out that this problem was documented in [14].

The possibility to actually execute a program in the Certicartes JCVM was another great benefit. Testing the transformations before starting the proofs helps to detect dummy bugs in the transformation.

Some parts of this proofs seems to be mechanical and very tedious to do manually. In order to speed up the proof a greater automation support is needed. This can be achieved by using more powerful tactics or using a different formal tool, for example a theorem prover could be used.

7. Future Work

Since we only complete part of the proof there is still a lot of work to do. In order to finish the proof we must prove the ten axioms of layer 1 and finish the proofs for layer 2 and 3. Through this article we have mentioned various works to do in order to extend the work done in this article; here is a summary of them:

- We also should prove that eliminating the `nop` instructions we have the same Java semantics and we should formalize the proof of BCV acceptance.
- Remove the restriction imposed at the equivalence theorem that the frame at the top of the stack must be executing over the modified method.
- Remove the restriction that a subroutine can only be called one time in a given method execution.
- Extend the instruction set with `tableswitch` or `lookupswitch` instructions

In addition to finishing the proofs, it would be interesting to extend the transformation and the proof to arbitrary subroutines. Another interesting challenge would be to reach higher levels of automation in the proofs by evaluating other formal tools or creating helpful COQ tactics.

8. References

[1] The Costs and Benefits of Java Bytecode Subroutines
Stephen N. Freund
Department of Computer Science, Stanford University, 1998

[2] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. The Java Series. Addison-Wesley, 1999. Second edition.

[3] Simple Verification Technique for Complex Java Bytecode Subroutines
Alessandro Coglio
Kestrel Institute, 2001

[4] Java bytecode verification: an overview
Xavier Leroy
INRIA Rocquencourt and Trusted Logic S.A.

[5] On-card Bytecode Verification for Java Card
Xavier Leroy
Trusted Logic S.A.

[6] Principles of Program Analysis
N. Flemming, N. Henne Riis, H. Chris
Springer 1999

[7] KVM Porting Guide
Sun Microsystems

[8] A Formal Executable Semantics of Java Card: Defensive and Offensive Virtual Machines
G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Melo de Sousa
INRIA Sophia-Antipolis, France

[9] The Problem of Bytecode Verification in Current Implementations of the JVM
Robert F. Stärk and Joachim Schmid

[10] The Coq Proof Assistant User's Guide. Version 6.3.1
B.Beckert, S.Boution, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, P. Loiseleur, C. Muños, C. Murthy, C. Parent-Vigouroux, C. Paulin-Mohring, A. Saïbi, and B. Werner
December 1999.

[11] The Coq Proof Assistant
http://coq.inria.fr

[12] Bytecode Model Checking: An Experimental Analysis
D. Basin, S. Friedrich, M. Gawkowski, J. Posegga
Proceedings of 9th International SPIN Workshop (ETAPS 2002)
Grenoble, France, April 2002.

[13] Java Bytecode verification using model checking
J. Posegga and H. Vogt
In the Workshop Fundamental Underpinnings of Java, 1998

[14] Java and the Java Virtual Machine
R. Stärk
Springer, 2001

[15] A Type System for Java Bytecode Subroutines
Raymie Stata, Martin Abadi