Specification and verification of a simple machine

Hanbing liu

March 8, 2006

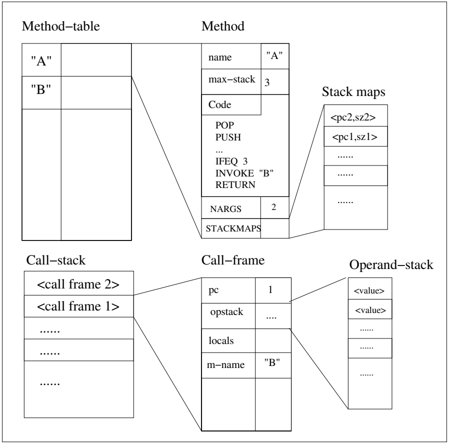
One slide summary

- Modeled a simple virtual machine: an interpreter + a static checker
- Proved that verified programs never overflow the operand stack
  - Identified a suitable “good-state” predicate
  - Proved that the “good-state” predicate is an inductive invariant of executing “verified” programs
  - Proved that a “good-state”’s operand stack is not too big

The proof input is 11,360 lines in 47 files. The machine model (the interpreter and the static checker) is just 913 lines.

Machine State

- State: call stack + method table
- Call frame: pc + operand-stack + locals + method-name
- Method: method-name + max-stack + code + nargs



Relevant concepts: current frame, current method, operand stack, current max-stack

Semantics of Instructions

- (PUSH `V`): push value `V` onto the current operand stack. Effects are undefined:
  - if the push will overflow the operand stack.
  - if the current frame does not exist ...
- (INVOKE `method-name`): look up method, initialize new frame, adjust old frame. Effects are undefined,
  - if there is not enough values on the operand stack
  - ...
- (RETURN):
  - effects are undefined, if the value returned will overflow the caller's operand stack
  - ....

Semantics of Instructions

- (PUSH `V`): push value `V` onto the current operand stack. Effects are undefined:
  - if the push will overflow the operand stack.
  - if the current frame does not exist ...
- (INVOKE `method-name`): look up method, initialize new frame, adjust old frame. Effects are undefined,
  - if there is not enough values on the operand stack
  - ...
- (RETURN):
  - effects are undefined, if the value returned will overflow the caller's operand stack
  - ....

This specification is incomplete unless one can prove that these "if" scenarios never arise. The official JVM specification defines the semantics of the instructions in the similar fashion.

What is expected of a specification

- Implementers
  - Prefer operationally specified system
- Application programmers
  - Need a complete specification
- End users
  - Want a complete specification.
  - Want a correct and efficient implementation.

What is expected of a specification

- Implementers
  - Prefer operationally specified system
- Application programmers
  - Need a complete specification
- End users
  - Want a complete specification.
  - Want a correct and efficient implementation.

As the specification designers, we want a complete and operationally specified specification. We want to design a virtual machine that can be implemented efficiently

Static checker

Objective: detect potentially *unsafe* programs before executing them.

Static checker

Objective: detect potentially *unsafe* programs before executing them.

High level view of the static checker:

- The specification demands that each method carries: code + "proof".
- The checker checks the "proof" against the code in the method
- If the checker accepts the "proof", the method is permitted for execution.

Static checker

Objective: detect potentially *unsafe* programs before executing them.

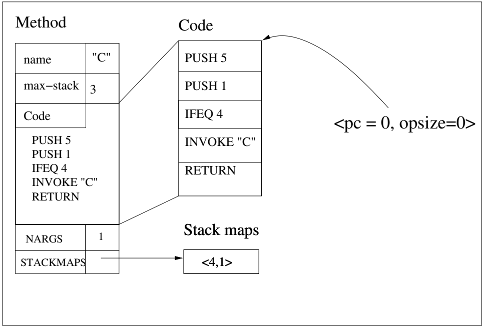
High level view of the static checker:

- The specification demands that each method carries: code + "proof".
- The checker checks the "proof" against the code in the method
- If the checker accepts the "proof", the method is permitted for execution.

The difficult task is to design a static checker and to prove it is *effective*.

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

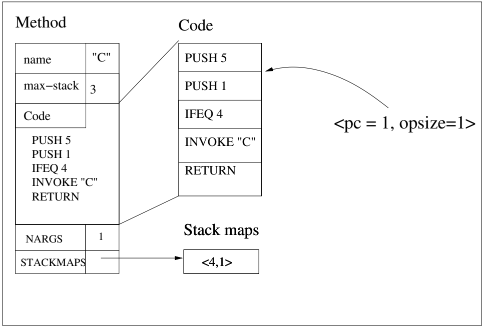


Check

- (a) opsize $= 0+1 < 3$ = max-stack
- (b) next pc in range

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

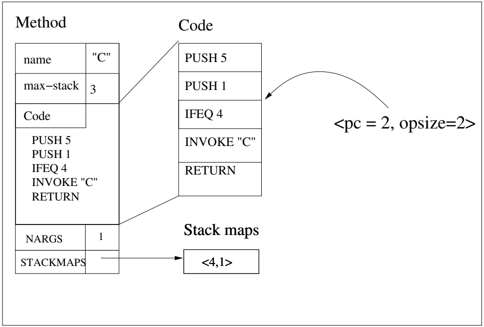


Check

- (a) opsize $= 1{+}1 < 3$ $=$ max-stack
- (b) next pc in range

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

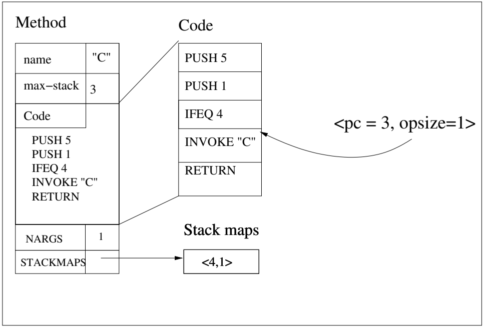


Check

- (a) opsize - 1 $<$ 3 $=$ max-stack
- (b.1) IFEQ target in range
- (b.2) opsize $=$ 2-1 $=$ 1 $=$ stackmap(4)
- (c) next pc in range

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

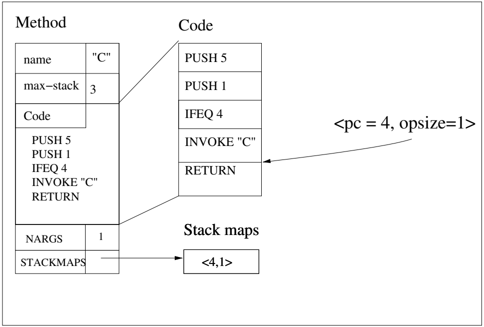


Check

- (a) opsize = 1 >= 1 = nargs
- (b) next pc in range
- (c) opsize -1 +1 = stackmap(4)
- (d) opsize -1 +1 <= max-stack

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

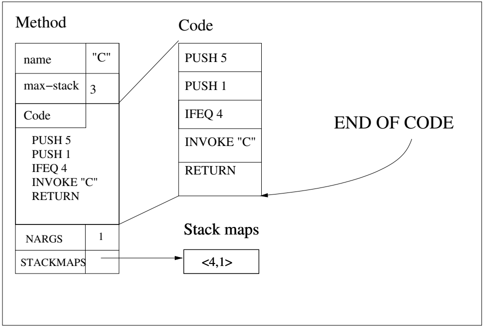


Check

- (a) opsize = 1 = stackmap(4)
- (b)
  - either next pc in range and stackmap(npc) defined
  - or there is no more code
- (c) enough operands

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

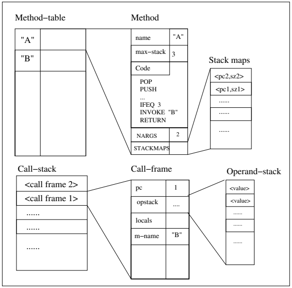


Verified!

- Success

State representation

State representation: use the `misc/record` book.



- (current-frame s) = (top (g 'call-stack s))
- (current-method s) = (binding (g 'method-name (current-frame s)) (g 'method-table s))
- Max stack: (max-stack s) = (g 'max-stack (current-method s))

State transition functions

State transition functions:

```
(defun djvm-check-INVOKE (inst st)
  (let* ((method-name  (arg inst))
         (method-table (g 'method-table st))
         (method    (binding method-name method-table))
         (nargs     (g 'nargs method)))
    (and (consistent-state st)
         (bound? method-name method-table)
         (<= 0 (g 'max-stack
                  (binding method-name method-table)))
         (integerp nargs)
         (<= 0 nargs)
         (<= nargs (len (op-stack st)))
         (<= (+ 1 (- (len (op-stack st))
                     nargs))
             (g 'max-stack (topx (g 'call-stack st))))
         (pc-in-range (set-pc (+ 1 (get-pc st))
                              st)))))
```

```
(defun execute-INVOKE (inst st)
  (let* ((method-name  (arg inst))
         (method-table (g 'method-table st))
         (method    (binding method-name
                             method-table))
         (nargs     (g 'nargs method)))
    (pushInitFrame
     method-name
     (init-locals (op-stack st) nargs)
     (set-pc (+ 1 (get-pc st))
             (popStack-n st nargs)))))
```

Static Checker Model

The static checker implementation follows the JVM bytecode verifier's specification

```
(defun bcv-method (method method-table)
  (let* ((code (g 'code method))
         (maps (g 'stackmaps method)))
    (and (wff-code (parsecode code))
         (wff-maps maps)
         (merged-code-safe
           (mergeStackMapAndCode
               maps
              (parsecode code)
              (g 'method-name method)
             method-table)
           (sig-method-init-frame method
                                  method-table)))))
```

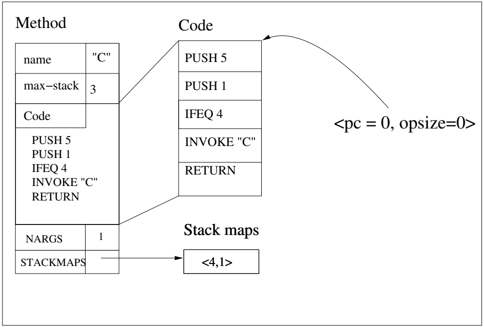
Static checker

Recall the high level view of the static checker as "proof checkers". The "proof checking" algorithm merges the stack maps and code into a sequence. It then symbolically executes the sequence.

- Before executing an instruction, the algorithm checks whether it is safe to execute the instruction.
- When encounters a stack map, the algorithm matches the current abstract state against the stack map.
- The algorithm accepts the code, if the symbolic execution reaches the end of the sequence without error.

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

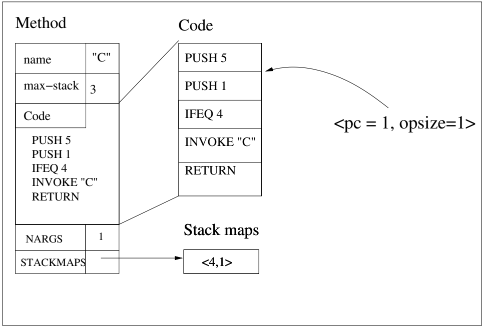


Check

- (a) opsize = 0+1 < 3 = max-stack
- (b) next pc in range

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: $<$pc, opsize$>$.

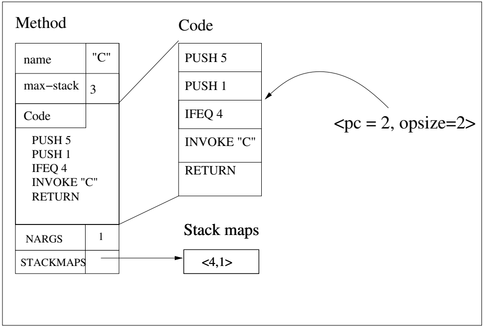


Check

- (a) opsize $= 1+1 < 3$ $=$ max-stack
- (b) next pc in range

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

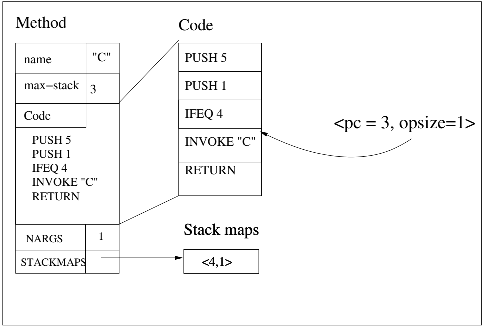


Check

- (a) opsize - 1 $<$ 3 $=$ max-stack
- (b.1) IFEQ target in range
- (b.2) opsize $=$ 2-1 $=$ 1 $=$ stackmap(4)
- (c) next pc in range

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

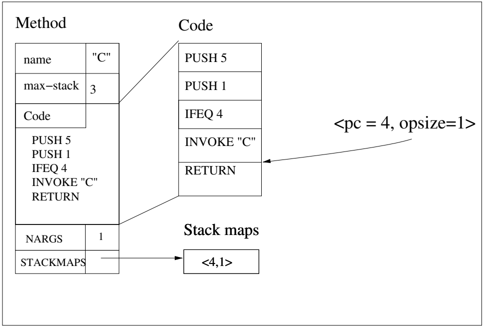


Check

- (a) opsize = 1 >= 1 = nargs
- (b) next pc in range
- (c) opsize -1 +1 = stackmap(4)
- (d) opsize -1 +1 <= max-stack

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.

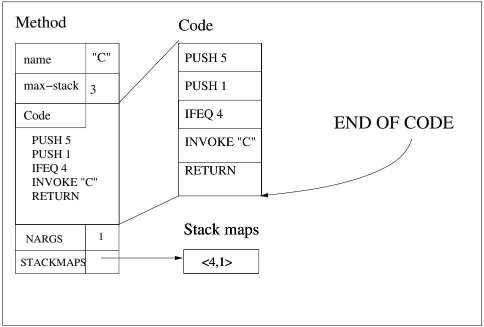


Check

- (a) opsize = 1 = stackmap(pc)
- (b)
  - either next pc in range and stackmap(npc) defined
  - or there is no more code

Static checker: algorithm

Static checker executes the method symbolically, maintaining an abstract state: <pc, opsize>.



Verified!

- Success

Why static checking works?

For PUSH and POP, the static checker's execution approximates the actual execution.

Why static checking works?

For PUSH and POP, the static checker's execution approximates the actual execution.

Static checker's executions diverge from the concrete executions when the static checker encounters IFEQ, INVOKE, RETURN.

- Static checker never takes the branch.
- Static checker assumes that INVOKE always returns.
- Static checker executes past RETURN

Why static checking works?

For PUSH and POP, the static checker's execution approximates the actual execution.

Static checker's executions diverge from the concrete executions when the static checker encounters IFEQ, INVOKE, RETURN.

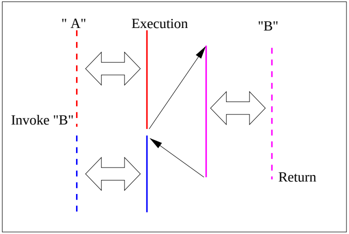
- Static checker never takes the branch.
- Static checker assumes that INVOKE always returns.
- Static checker executes past RETURN

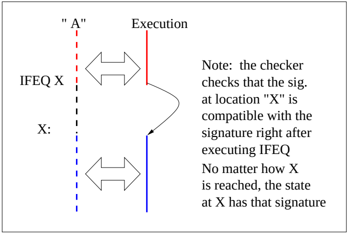
The static checker demands that the stackmap are provided at the branch targets (and immediately after the RETURN as well).
Intuition is:

Why static checking works?

Intuition:

- Any concrete execution can be “chopped” into segments.
- Executing a verified program, every segment is approximated with some segment from the static checker execution on the program.

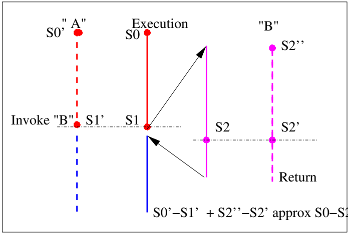




How to formalize it

Either:

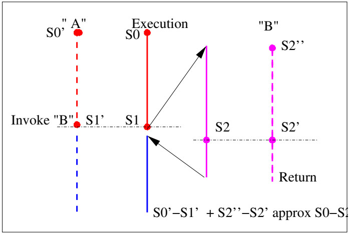
- Formalize the idea of *trace* and *segments* explicitly



How to formalize it

Either:

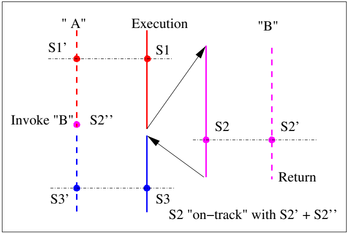
- Formalize the idea of *trace* and *segments* explicitly



- Or, formalize the concept that a machine state is "on-track" with some static checker's execution

Consistent state

Concept of “on-track”



The machine state has a call stack that records the execution history upto now.

- Each caller’s call frame corresponds to some “unfinished” execution of some subprogram.

Approach

Observations:

- The original static checker returns “yes” and “no”
- We need the intermediate state of symbolic simulation to state the “on-track” properties.

Approach

Observations:

- The original static checker returns "yes" and "no"
- We need the intermediate state of symbolic simulation to state the "on-track" properties.

Rough solution ideas:

- Imagine an "observer" X that monitors the static checker run and records the intermediate states that static checker encountered.
- State "on-track" property.
- Prove when the static checker succeeds, "on-track" property is preserve.
- Prove when "on-track" is true and the static checker succeeds, effects of executing machine operations are well defined.

Extra complications

Observations:

- The original static checker returns "yes" and "no" for the whole program
- However, we are proving step-wise properties: (1) "on-track" is preserved (2) when "on-track", it is safe to execute a step. We need to reason about the corresponding small step taken by the static checker.

Extra complications

Rough solution ideas:

- A simpler checker which expects that every instruction is annotated with stackmaps.
- The simpler checker checks that for every instruction, it is safe execute the instruction under the specified context and the resulting states of executing the instruction are compatible with annotations.
- Prove the machine is "on-track" with this simpler checker.
- Prove if the original static checker succeeds, the observer X generates annotations that makes the simpler checker succeed.

Static checking is effective

"Progress"

```
(defthm djvm-check-succeed-in-consistent-state
  (implies (and (CONSISTENT-STATE DJVM-S)
                (bcv-verified-method-table
                          (g 'method-table djvm-s)))
           (djvm-check-step djvm-s)))
```

Static checking is effective

"Preservation"

```
(defthm djvm-step-preserve-consistent-state
   (implies (consistent-state st)
            (consistent-state (djvm-step st))))
```

Static checking is effective

“The static checker allows efficient implementation”

```
(defthm verified-program-executes-safely
  (implies (and (consistent-state djvm-s)
                (state-equiv jvm-s djvm-s)
                (bcv-verified-method-table
                             (g 'method-table djvm-s)))
           (state-equiv (m-run jvm-s n)
                        (djvm-run djvm-s n))))
```

Static checking is effective

"Verified code never overflow operand stack"

```
(defthm verified-program-never-overflow-operand-stack-in-jvm
  (implies
    (and (consistent-state djvm-s)
         (state-equiv jvm-s djvm-s)
         (all-method-verified (g 'method-table djvm-s)))
  (<= (len (g 'op-stack (topx (g 'call-stack (m-run jvm-s n)))))
      (max-stack
       (binding
        (g 'method-name (topx (g 'call-stack (m-run jvm-s n))))
        (g 'method-table jvm-s))))))
```