Mechanized Operational Semantics

J Strother Moore
Department of Computer Sciences
University of Texas at Austin

Marktoberdorf Summer School 2008

(Lecture 3: Direct Proofs)

Fact 1

Given an operational semantics, *symbolic execution* of code is just substitution-of-equals-for-equals, i.e., rewriting.

```
'((PUSH 1)     ;  0
  (STORE 1)    ;  1    a := 1
  (LOAD 0)     ;  2 loop
  (IFLE 10)    ;  3    if n<=0 go end
  (LOAD 0)     ;  4
  (LOAD 1)     ;  5
  (MUL)        ;  6
  (STORE 1)    ;  7    a := n*a
  (LOAD 0)     ;  8
  (PUSH 1)     ;  9
  (SUB)        ; 10
  (STORE 0)    ; 11    n := n-1
  (GOTO -10)   ; 12    go loop
  (LOAD 1)     ; 13 end
  (RETURN)))   ; 14    return a
```

```
'((PUSH 1)     ;  0
  (STORE 1)    ;  1    a := 1
  (LOAD 0)     ;  2 loop
  (IFLE 10)    ;  3    if n<=0 go end
  (LOAD 0)     ;  4
  (LOAD 1)     ;  5
  (MUL)        ;  6
  (STORE 1)    ;  7    a := n*a
  (LOAD 0)     ;  8
  (PUSH 1)     ;  9
  (SUB)        ; 10
  (STORE 0)    ; 11    n := n-1
  (GOTO -10)   ; 12    go loop
  (LOAD 1)     ; 13 end
  (RETURN)))   ; 14    return a
```

```
(run (repeat 0 9)
     (make-state 4
                 (list n a)
                 stk
                 *g*))
```

```
(run '(0 0 0 0 0 0 0 0 0 0)
     (make-state 4
                 (list n a)
                 stk
                 *g*))
```

```
(run '(0 0 0 0 0 0 0 0)
     (step
      (make-state 4
                  (list n a)
                  stk
                  *g*)))
```

```
(run '(0 0 0 0 0 0 0)
     (step
      (step
       (make-state 4
                   (list n a)
                   stk
                   *g*))))
```

```
(run '(0 0 0 0 0 0)
     (step
      (step
       (step
        (step
         (make-state 4
                     (list n a)
                     stk
                     *g*)))))
```

```
(run '(0 0 0 0 0)
     (step
      (step
       (step
        (step
         (step
          (make-state 4
                      (list n a)
                      stk
                      *g*))))))
```

```
(run '(0 0 0 0)
     (step
      (step
       (step
        (step
         (step
          (step
           (make-state 4
                       (list n a)
                       stk
                       *g*)))))))
```

```
(run '(0 0 0)
     (step
      (step
       (step
        (step
         (step
          (step
           (step
            (make-state 4
                        (list n a)
                        stk
```

```
(run '(0 0)
     (step
      (step
       (step
        (step
         (step
          (step
           (step
            (step
             (make-state 4
                         (list n a)
```

```
(run '(0)
     (step
      (step
       (step
        (step
         (step
          (step
           (step
            (step
             (step
              (make-state 4
```

```
(run '()
     (step
      (step
       (step
        (step
         (step
          (step
           (step
            (step
             (step
              (step
```

```
(step
 (step
  (step
   (step
    (step
     (step
      (step
       (step
        (step
         (step
          (make-state 4 ...
```

Consider

```
(step (make-state 4 (list n a) stk *g*))
```

```
(defun step (s) (do-inst (next-inst s) s))

(defun next-inst (s) (nth (pc s) (program s)))

(defconst *g*
  '((PUSH 1)     ;  0
    (STORE 1)    ;  1
    (LOAD 0)     ;  2
    (IFLE 10)    ;  3
    (LOAD 0)     ;  4
```

Consider

```
(step (make-state 4 (list n a) stk *g*))
=
(execute-LOAD '(LOAD 0)
    (make-state 4 (list n a) stk *g*))
```

```
(defun execute-LOAD (inst s)
  (make-state (+ 1 (pc s))
              (locals s)
              (push (nth (arg1 inst)
                         (locals s))
                    (stack s))
              (program s)))
```

Consider

```
(step (make-state 4 (list n a) stk *g*))
=
(execute-LOAD '(LOAD 0)
    (make-state 4 (list n a) stk *g*))
```

Consider

```
(step (make-state 4 (list n a) stk *g*))
=
(execute-LOAD '(LOAD 0)
    (make-state 4 (list n a) stk *g*))
=
(make-state 5
            (list n a)
            (push n stk)
            *g*)
```

```
(step
 (step
  (step
   (step
    (step
     (step
      (step
       (step
        (step
         (step
          (make-state 4 ;(LOAD 0)
```

```
(step
 (step
  (step
   (step
    (step
     (step
      (step
       (step
        (step
         (make-state 5 ;(LOAD 1)
                     (list n a)
```

```
(step
 (step
  (step
   (step
    (step
     (step
      (step
       (step
        (make-state 6 ;(MUL)
                    (list n a)
                    (push a (push n stk))
```

```
(step
 (step
  (step
   (step
    (step
     (step
      (make-state 7 ;(STORE 1)
                  (list n a)
                  (push (* a n) stk)
                  *g*)))))))
```

```
(step
 (step
  (step
   (step
    (step
     (step
       (make-state 8 ;(LOAD 0)
                     (list n (* a n))
                     stk
                     *g*))))))
```

```
(step
 (step
  (step
   (step
     (make-state 9 ;(PUSH 1)
                   (list n (* a n))
                   (push n stk)
                   *g*)))))
```

```
(step
 (step
  (step
    (make-state 10 ;(SUB)
                (list n (* a n))
                (push 1 (push n stk))
                *g*))))
```

```
(step
 (step
   (make-state 11 ;(STORE 0)
               (list n (* a n))
               (push (- n 1) stk)
               *g*)))
```

```
(step
  (make-state 12 ;(GOTO -10)
              (list (- n 1) (* a n))
              stk
              *g*))
```

```
(make-state 2
            (list (- n 1) (* a n))
            stk
            *g*)
```

Fact 1

Given an operational semantics, *symbolic execution* of code is just substitution-of-equals-for-equals, i.e., rewriting.

Demo 1

Theorem?

```
(equal (run (repeat 0 9)
            (make-state 4
                        (list n a)
                        stk
                        *g*))
       ???)
```

Fact 2

`Append` (of schedules) is just *sequential composition*.

Theorem. `run-append`

```
(equal (run (append a b) s)
       (run b (run a s)))
```

Demo 2

Aside

Note that a virtue of having an operational semanics expressed in a formal logic is that we can prove theorems *about the semantics*, independent of any particular *program*.

Having proved `run-append`, whenever the system encounters:

`(run (append` α β`)` σ`)`

it will replace it by

`(run` β `(run` α σ`))`

```
(defun g-sched-loop (n)
  (if (zp n)
      (repeat 0 4)
    (append (repeat 0 11)
            (g-sched-loop (- n 1)))))
```

Suppose `(zp n)` is known to be false.

`(run (g-sched-loop n)` σ`)`

Suppose `(zp n)` is known to be false.

```
(run (g-sched-loop n) σ)
=
(run (append (repeat 0 11)
             (g-sched-loop (- n 1)))
     σ)
```

Suppose (zp n) is known to be false.

```
(run (g-sched-loop n) σ)
=
(run (append (repeat 0 11)
             (g-sched-loop (- n 1)))
     σ)
=
(run (g-sched-loop (- n 1))
     (run (repeat 0 11) σ))
```

To Prove Code Correct

Proceed in two steps:

- prove code implements algorithm – attack innermost loop first
- prove algorithm implements specification – straight mathematical proof, no code or operational semantics involved

Example

Let us prove that `*g*` computes factorial.

```
(defun ! (n)
  (if (zp n)
      1
    (* n (! (- n 1)))))
```

Example

```
(equal (top
        (stack
         (run (g-sched n)
              (make-state 0
                          (list n a)
                          stk
                          *g*))))
        (! n))
```

Better Example!

```
(equal (run (g-sched n)
            (make-state 0
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (! n))
                   (push (! n) stk)
                   *g*))
```

Aside

Proving stronger theorems is often easier when induction is used.

The complete characterization of the effects of *g* are essential for $security$ proofs.

Step 1: Code Implements Algorithm

```
(defun g (n a)
  (if (zp n)
      a
    (g (- n 1) (* n a))))
```

In the following, we will assume `n` and `a` are natural numbers.

Step 1: The Loop Lemma

Suppose the locals are `n` and `a`.

Suppose you start at the top of the loop (`pc` $= 2$) and run the schedule for the loop (`(g-sched-loop n)`).

What is the final state?

Step 1: The Loop Lemma

```
(equal (run (g-sched-loop n)
            (make-state 2
                        (list n a)
                        stk
                        *g*))
       ???)
```

Step 1: The Loop Lemma

```
(equal (run (g-sched-loop n)
            (make-state 2
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n a))
                   (push (g n a) stk)
                   *g*))
```

Proof

Induct on `n` (as suggested by `(g n a)`)

Base Case:

`(implies (zp n)` ψ`(n,a))`

Induction Step:

```
(implies (and (not (zp n))
              ψ((- n 1), (* a n)))
         ψ(n,a))
```

Base Case: (zp n)

```
(equal (run (g-sched-loop n)
            (make-state 2
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n a))
                   (push (g n a) stk)
                   *g*))
```

Base Case: (zp n)

```
(equal (run (repeat 0 4)
            (make-state 2
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n a))
                   (push (g n a) stk)
                   *g*))
```

Base Case: (zp n)

```
(equal (run (repeat 0 4)
            (make-state 2
                        (list 0 a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g 0 a))
                   (push (g 0 a) stk)
                   *g*))
```

Base Case: (zp n)

```
(equal (run (repeat 0 4)
            (make-state 2
                        (list 0 a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 a)
                   (push a stk)
                   *g*))
```

```
'((PUSH 1)     ;  0
  (STORE 1)    ;  1    a := 1
  (LOAD 0)     ;  2 loop
  (IFLE 10)    ;  3    if n<=0 go end
  (LOAD 0)     ;  4
  (LOAD 1)     ;  5
  (MUL)        ;  6
  (STORE 1)    ;  7    a := n*a
  (LOAD 0)     ;  8
  (PUSH 1)     ;  9
  (SUB)        ; 10
  (STORE 0)    ; 11    n := n-1
  (GOTO -10)   ; 12    go loop
  (LOAD 1)     ; 13 end
  (RETURN)))   ; 14    return a
```

Base Case: (zp n)

```
(equal (run (repeat 0 4)
            (make-state 2
                        (list 0 a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 a)
                   (push a stk)
                   *g*))
```

Base Case: (zp n)

```
(equal (make-state 14
                   (list 0 a)
                   (push a stk)
                   *g*)
       (make-state 14
                   (list 0 a)
                   (push a stk)
                   *g*))
```

Base Case: `(zp n)`

`T`

Induction Conclusion [(not (zp n))]

```
(equal
 (run (g-sched-loop n)
      (make-state 2
                  (list n a)
                  stk
                  *g*))
 (make-state 14
             (list 0 (g n a))
             (push (g n a) stk)
             *g*))
```

Induction Hypothesis:

```
(equal
 (run (g-sched-loop (- n 1))
      (make-state 2
                  (list (- n 1) (* a n))
                  stk
                  *g*))
 (make-state 14
             (list 0 (g (- n 1) (* a n)))
             (push (g (- n 1) (* a n)) stk)
             *g*))
```

Induction Conclusion [(not (zp n))]

```
(equal
 (run (g-sched-loop n)
      (make-state 2
                  (list n a)
                  stk
                  *g*))
 (make-state 14
             (list 0 (g n a))
             (push (g n a) stk)
             *g*))
```

Induction Conclusion [(not (zp n))]

```
(equal
 (run (append (repeat 0 11)
              (g-sched-loop (- n 1)))
      (make-state 2
                  (list n a)
                  stk
                  *g*))
 (make-state 14
             (list 0 (g n a))
             (push (g n a) stk)
```

Induction Conclusion [(not (zp n))]

```
(equal
 (run (g-sched-loop (- n 1))
      (run (repeat 0 11)
           (make-state 2
                       (list n a)
                       stk
                       *g*)))
 (make-state 14
             (list 0 (g n a))
             (push (g n a) stk)
```

Induction Conclusion [(not (zp n))]

```
(equal
 (run (g-sched-loop (- n 1))
      (run (repeat 0 11)
           (make-state 2
                       (list n a)
                       stk
                       *g*)))
 (make-state 14
             (list 0 (g n a))
             (push (g n a) stk)
```

Induction Conclusion [(not (zp n))]

```
(equal
 (run (g-sched-loop (- n 1))
      (make-state 2
                  (list (- n 1) (* a n))
                  stk
                  *g*))
 (make-state 14
             (list 0 (g n a))
             (push (g n a) stk)
             *g*))
```

Induction Conclusion [(not (zp n))]

```
(equal
 (run (g-sched-loop (- n 1))
      (make-state 2
                  (list (- n 1) (* a n))
                  stk
                  *g*))
 (make-state 14
             (list 0 (g (- n 1) (* a n)))
             (push (g (- n 1) (* a n)) stk)
             *g*))
```

Induction Hypothesis:

```
(equal
 (run (g-sched-loop (- n 1))
      (make-state 2
                  (list (- n 1) (* a n))
                  stk
                  *g*))
 (make-state 14
             (list 0 (g (- n 1) (* a n)))
             (push (g (- n 1) (* a n)) stk)
             *g*))
```

Induction Conclusion [(not (zp n))]

```
(equal
 (run (g-sched-loop (- n 1))
      (make-state 2
                  (list (- n 1) (* a n))
                  stk
                  *g*))
 (make-state 14
             (list 0 (g (- n 1) (* a n)))
             (push (g (- n 1) (* a n)) stk)
             *g*))
```

Induction Conclusion [`(not (zp n))`]

`T`

Q.E.D.

Demo 3

Step 1: The Main Code Theorem

If you run the full schedule, `(g-sched n)`, starting at the main entry ($\texttt{pc} = 2$), the code computes `(g n 1)`.

Step 1: The Main Code Theorem

```
(equal (run (g-sched n)
            (make-state 0
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
                   *g*))
```

Step 1: The Main Code Theorem

```
(equal (run (g-sched n)
            (make-state 0
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
                   *g*))
```

Step 1: The Main Code Theorem

```
(equal (run (append (repeat 0 2)
                    (g-sched-loop n))
            (make-state 0
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
```

Step 1: The Main Code Theorem

```
(equal (run (g-sched-loop n)
            (run (repeat 0 2)
                 (make-state 0
                             (list n a)
                             stk
                             *g*)))
       (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
```

Step 1: The Main Code Theorem

```
(equal (run (g-sched-loop n)
            (run (repeat 0 2)
                 (make-state 0
                             (list n a)
                             stk
                             *g*)))
       (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
```

Step 1: The Main Code Theorem

```
(equal (run (g-sched-loop n)
            (make-state 2
                        (list n 1)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
                   *g*))
```

Step 1: The Loop Lemma

```
(equal (run (g-sched-loop n)
            (make-state 2
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n a))
                   (push (g n a) stk)
                   *g*))
```

Step 1: The Main Code Theorem

```
(equal (run (g-sched-loop n)
            (make-state 2
                        (list n 1)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
                   *g*))
```

Step 1: The Main Code Theorem

```
(equal (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
                   *g*)
       (make-state 14
                   (list 0 (g n 1))
                   (push (g n 1) stk)
                   *g*))
```

Step 1: The Main Code Theorem

T

Q.E.D.

Demo 4

Step 2

So we have proved that the code implements the algorithm. It remains to show that the algorithm implements the specification.

That is, we have to prove that `g` "is" `f`.

Step 2: Prove Algorithm Implements Spec

```
(defthm step-2
  (implies (natp a)
           (equal (g n a)
                  (* a (! n)))))
```

Main Theorem

```
(equal (run (g-sched n)
            (make-state 0
                        (list n a)
                        stk
                        *g*))
       (make-state 14
                   (list 0 (! n))
                   (push (! n) stk)
                   *g*))
```

Corollary

```
(let ((s_fin (run (g-sched n)
                  (make-state 0
                              (list n a)
                              stk
                              *g*))))
  (implies (natp n)
           (and (equal (top (stack s_fin))
                       (! n))
                (haltedp s_fin))))
```

Demo 5

For many high-level languages, the semantics is given only by the compiler!

Verifying the output of the compiler means you do not have to trust the compiler.

You are verifying what actually executes.

Consider Berkeley C String Library and gcc.

Next Time

Conventional inductive invariant proofs