CS378 - A Formal Model of the JVM
Lectures 6,7,8

http://www.cs.utexas.edu/users/moore/classes/cs378-jvm/

Semester	Spring, 2012
Unique Id	53105
Intructor	J Strother Moore
Email	moore@cs.utexas.edu
Office	MAI 2014
Office Hours	MW 1:00–2:00

Review M1 Release on Web

Two Flavors of Correctness

Partial:
If computation on ok inputs halts, then the answer is as expected.

Total:
Computation from ok inputs halts and the answer is as expected.

State-Based Formalization

The most general way to formulate these conditions is in terms of entire M1 states.

`(pre` s_i`)` – checks that s_i is an acceptable initial state for your program

`(post` s_i s_f`)` – checks that the final state, s_f, is in the expected relation with the initial state s_i

(`halted` s_f) – checks that state s_f is halted, e.g., (`next-inst` s_f) = `'(HALT)`

Most-General State Based Partial Correctness

$\forall\ s_i, \sigma$:

((pre s_i)

$\wedge$

(halted (run σ s_i))

$\rightarrow$

(post s_i (run σ s_i))

Most-General State Based Total Correctness

$\forall\ s_i\ \exists\ \sigma$:

(pre s_i)

$\rightarrow$

((halted (run σ s_i))

$\wedge$

(post s_i (run σ s_i)))

However...

It is generally more convenient to relate program inputs and outputs rather than whole states.

So for the next few lectures we deal with slightly less general notions of correctness...

Typical Idioms in M1 Correctness Statements

- the initial state, sometimes called s_i and often written

  `(make-state 0 (list` $v_1 \ldots$`) nil` π`)`

- the acceptable values of the inputs (local variables), $v_1, \ldots$, e.g., `(natp` v_1`)`

- the schedule, σ, sometimes known and sometimes not

- the final state, s_f, usually just `(run` σ s_i`)`
- where in the final state the answer is found, e.g., `(top (stack` s_f`))`
- the expected answer $\theta_{v_1,\ldots}$, i.e., some expression in terms of the inputs, e.g., `(*` v_1 v_2`)`.

Partial Correctness

"If computation on ok inputs halts, then the answer is as expected."

$$
\begin{array}{l}
(\quad s_i = \texttt{(make-state 0 (list } v_1 \ldots\texttt{) nil } \pi\texttt{)} \\
\;\wedge\; \texttt{(ok-inputs } v_1 \ldots\texttt{)} \\
\;\wedge\; s_f = \texttt{(run sched } s_i\texttt{)} \\
\;\wedge\; \texttt{(next-inst } s_f\texttt{)} = \texttt{'(HALT))} \\
\rightarrow \\
\;\texttt{(top (stack } s_f\texttt{))} = \theta\texttt{.}
\end{array}
$$

Partial Correctness

"If computation on ok inputs halts, then the answer is as expected."

$$
\begin{array}{l}
(\quad s_i = (\texttt{make-state 0 (list } v_1 \ldots\texttt{) nil } \pi\texttt{)} \\
\;\wedge\; (\texttt{ok-inputs } v_1 \ldots) \\
\;\wedge\; s_f = (\texttt{run sched } s_i) \\
\;\wedge\; (\texttt{next-inst } s_f) = \texttt{'(HALT))} \\
\rightarrow \\
(\texttt{top (stack } s_f\texttt{))} = \theta.
\end{array}
$$

You may eliminate equality hypotheses that just "name" expressions.

Partial Correctness

$$(\quad s_i = \texttt{(make-state 0 (list } v_1 \ldots\texttt{) nil } \pi\texttt{)}$$
$$\wedge \texttt{ (ok-inputs } v_1 \ldots\texttt{)}$$
$$\wedge\ s_f = \texttt{(run sched } s_i\texttt{)}$$
$$\wedge \texttt{ (next-inst } s_f\texttt{) = '(HALT))}$$
$$\rightarrow$$
$$\texttt{(top (stack } s_f\texttt{)) = } \theta\texttt{.}$$

Partial Correctness

$$
\begin{array}{l}
(\\
\quad \wedge\ (\texttt{ok-inputs}\ v_1 \ldots)\\
\quad \wedge\ s_f = (\texttt{run sched (make-state 0 (list}\ v_1 \ldots \texttt{)}\\
\quad \wedge\ (\texttt{next-inst}\ s_f) = \texttt{'(HALT))}\\
\rightarrow\\
(\texttt{top (stack}\ s_f\texttt{))} = \theta.
\end{array}
$$

Partial Correctness

$$
\begin{array}{l}
(\quad (\texttt{ok-inputs } v_1 \ldots) \\
\ \wedge\ s_f = (\texttt{run sched} \\
\qquad\qquad\qquad (\texttt{make-state 0 (list } v_1 \ldots\texttt{) nil } \pi)) \\
\ \wedge\ (\texttt{next-inst } s_f) = \texttt{'(HALT))} \\
\rightarrow \\
(\texttt{top (stack } s_f)) = \theta.
\end{array}
$$

Partial Correctness

$$
\begin{array}{l}
(\quad (\texttt{ok-inputs } v_1 \ldots) \\
\ \wedge\ s_f = (\texttt{run sched} \\
\qquad\qquad (\texttt{make-state 0 (list } v_1 \ldots\texttt{) nil } \pi\texttt{))} \\
\ \wedge\ (\texttt{next-inst } s_f) = \texttt{'(HALT))} \\
\rightarrow \\
(\texttt{top (stack } s_f\texttt{))} = \theta.
\end{array}
$$

Partial Correctness

```
(   (ok-inputs v1 ...)
 ∧ (next-inst
      (run sched
       (make-state 0 (list v1 ...) nil π)))
     = '(HALT))
→
 (top
  (stack
   (run sched
    (make-state 0 (list v1 ...) nil π))))
 = θ.
```

$$\begin{aligned}&(\ \ (\texttt{ok-inputs}\ v_1 \ldots)\\&\ \wedge (\texttt{next-inst}\ (\texttt{run sched}\ (\texttt{make-state 0 (list}\ v_1 \ldots\texttt{) nil}\ \pi)))\ = \texttt{'(HALT))}\\&\rightarrow\\&(\texttt{top}\ (\texttt{stack}\ (\texttt{run sched}\ (\texttt{make-state 0 (list}\ v_1 \ldots\texttt{) nil}\ \pi))))\\&= \theta.\end{aligned}$$

Partial Correctness

$$
\begin{array}{l}
(\quad s_i = \texttt{(make-state 0 (list } v_1 \ldots\texttt{) nil } \pi\texttt{)} \\
\ \wedge\ \texttt{(ok-inputs } v_1 \ldots\texttt{)} \\
\ \wedge\ s_f = \texttt{(run sched } s_i\texttt{)} \\
\ \wedge\ \texttt{(next-inst } s_f\texttt{)} = \texttt{'(HALT))} \\
\rightarrow \\
\texttt{(top (stack } s_f\texttt{))} = \theta.
\end{array}
$$

Partial Correctness (in ACL2)

(implies
 (and
  (equal s_i (make-state 0 (list $v_1 \ldots$) nil π))
  (ok-inputs $v_1 \ldots$)
  (equal s_f (run sched s_i))
  (equal (next-inst s_f) '(HALT)))
 (equal (top (stack s_f)) θ))

Total Correctness

"Computation from ok inputs halts and the answer is as expected."

$$
\begin{array}{l}
\exists\ \sigma\ \texttt{:} \\
\ \ (\quad s_i = (\texttt{make-state 0 (list } v_1 \ldots\texttt{) nil } \pi\texttt{)} \\
\ \ \ \ \wedge\ (\texttt{ok-inputs } v_1 \ldots) \\
\ \ \ \ \wedge\ s_f = (\texttt{run } \sigma\ s_i)) \\
\rightarrow \\
\ \ (\ (\texttt{next-inst } s_f) = \texttt{'(HALT)} \\
\ \ \ \ \wedge \\
\ \ \ \ \ \ (\texttt{top (stack } s_f\texttt{))} = \theta\texttt{).}
\end{array}
$$

Total Correctness (in ACL2)

"Computation from ok inputs halts and the answer is as expected."

$\exists\ \sigma$:

```
(implies
 (and
  (equal sᵢ (make-state 0 (list v₁...) nil π))
  (ok-inputs v₁...)
  (equal s_f (run σ sᵢ)))
 (and (equal (next-inst s_f) '(HALT))
      (equal (top (stack s_f)) θ)))
```

Total Correctness (in ACL2)

"Computation from ok inputs halts and the answer is as expected."

$\exists\ \sigma$:

```
(implies
 (and
  (equal s_i (make-state 0 (list v_1 ...) nil π))
  (ok-inputs v_1 ...)
  (equal s_f (run σ s_i)))
 (and (equal (next-inst s_f) '(HALT))
      (equal (top (stack s_f)) θ)))
```

Total Correctness (in ACL2)

"Computation from ok inputs halts and the answer is as expected."

```
(implies
 (and
  (equal s_i (make-state 0 (list v_1 ...) nil π))
  (ok-inputs v_1 ...)
  (equal s_f (run (sched v_1 ...) s_i)))
 (and (equal (next-inst s_f) '(HALT))
      (equal (top (stack s_f)) θ)))
```

Total Correctness

"Computation from ok inputs halts and the answer is as expected."

$$
\begin{array}{l}
\exists\ \sigma\ : \\
\ (\quad s_i\ =\ (\texttt{make-state 0 (list}\ v_1\ldots\texttt{) nil}\ \pi\texttt{)} \\
\quad \wedge\ \texttt{(ok-inputs}\ v_1\ldots\texttt{)} \\
\quad \wedge\ s_f\ =\ \texttt{(run}\ \sigma\ \ s_i\texttt{))} \\
\rightarrow \\
\ (\ \texttt{(next-inst}\ s_f\texttt{)}\ =\ \texttt{'(HALT)} \\
\quad \wedge \\
\qquad \texttt{(top (stack}\ s_f\texttt{))}\ =\ \theta\texttt{).}
\end{array}
$$

Total Correctness

"Computation from ok inputs halts and the answer is as expected."

$$
\begin{array}{l}
\exists\ \sigma\ :\\
\ (\quad s_i\ =\ \texttt{(make-state 0 (list } v_1\ldots\texttt{) nil } \pi\texttt{)}\\
\quad \wedge\ \texttt{(ok-inputs } v_1\ldots\texttt{)}\\
\quad \wedge\ s_f\ =\ \texttt{(run } \sigma\ s_i\texttt{))}\\
\rightarrow\\
\ (\ \texttt{(next-inst } s_f\texttt{)}\ =\ \texttt{'(HALT)}\\
\quad \wedge\\
\qquad \texttt{(top (stack } s_f\texttt{))}\ =\ \theta\texttt{)}.
\end{array}
$$

Bogus Correctness

"Computation from ok inputs produces the answer expected."

$$\exists\ \sigma\ :$$

$$\begin{array}{l}
(\quad s_i = (\texttt{make-state 0 (list } v_1 \ldots\texttt{) nil } \pi\texttt{)} \\
\ \wedge\ (\texttt{ok-inputs } v_1 \ldots\texttt{)} \\
\ \wedge\ s_f = (\texttt{run } \sigma\ s_i\texttt{))} \\
\rightarrow \\
\\
\quad (\texttt{top (stack } s_f\texttt{))} = \theta\texttt{.}
\end{array}$$

Demo 1

We Will Focus on Total Correctness

$\exists\ \sigma$:

```
 (   si = (make-state 0
                      (list v1 ...)
                      nil
                      π)
   ∧ (ok-inputs v1 ...)
   ∧ sf = (run σ si))
→
 ( (next-inst sf) = '(HALT)
   ∧
    (top (stack sf)) = θ).
```

We Will Focus on Total Correctness

$$
\begin{array}{l}
\exists\ \sigma\ : \\
\ (\quad s_i = \texttt{(make-state 0} \\
\qquad\qquad\qquad\qquad \texttt{(list } v_1 \ldots \texttt{)} \\
\qquad\qquad\qquad\qquad \texttt{nil} \\
\qquad\qquad\qquad\qquad \pi \texttt{)} \\
\quad \wedge\ \texttt{(ok-inputs } v_1 \ldots \texttt{)} \\
\quad \wedge\ s_f = \texttt{(run } \sigma\ s_i \texttt{))} \\
\rightarrow \\
\ (\ \texttt{(next-inst } s_f \texttt{)} = \texttt{'(HALT)} \\
\quad \wedge \\
\qquad \texttt{(top (stack } s_f \texttt{))} = \theta \texttt{).}
\end{array}
$$

We Will Focus on Total Correctness

$$
\begin{array}{l}
(\quad s_i = (\texttt{make-state}\ 0 \\
\qquad\qquad\qquad\qquad (\texttt{list}\ v_1 \ldots) \\
\qquad\qquad\qquad\qquad \texttt{nil} \\
\qquad\qquad\qquad\qquad \pi) \\
\quad \wedge\ (\texttt{ok-inputs}\ v_1 \ldots) \\
\quad \wedge\ s_f = (\texttt{run}\ (\texttt{sched}\ v_1 \ldots)\ s_i)) \\
\rightarrow \\
(\ (\texttt{next-inst}\ s_f) = \texttt{'(HALT)} \\
\quad \wedge \\
\quad (\texttt{top}\ (\texttt{stack}\ s_f)) = \theta).
\end{array}
$$

Recall

```
(defconst *g-program*
       '((ICONST 0)   ; 0
         (ISTORE 2)   ; 1  a = 0;
         (ILOAD 0)    ; 2  [loop:]
         (IFEQ 10)    ; 3  if x=0 then go to end;
         (ILOAD 0)    ; 4
         (ICONST 1)   ; 5
         (ISUB)       ; 6
         (ISTORE 0)   ; 7  x = x-1;
         (ILOAD 1)    ; 8
         (ILOAD 2)    ; 9
         (IADD)       ;10
         (ISTORE 2)   ;11  a = y+a;
         (GOTO -10)   ;12  go to loop
         (ILOAD 2)    ;13  [end:]
         (HALT)))     ;14 ``return'' a
```

Goal

Let's specify and prove the total correctness of `*g-program*`, namely, that the expected result is the product of the two inputs.

Total Correctness of *g-program*

```
(   s_i = (make-state 0
                      (list x y)
                      nil
                      *g-program*)
  ∧ (ok-inputs x y)
  ∧ s_f = (run (sched x y) s_i))
→
( (next-inst s_f) = '(HALT)
  ∧
  (top (stack s_f)) = (* x y)).
```

$$\begin{aligned}
&(\quad s_i = \texttt{(make-state 0 (list x y) nil *g-program*)}\\
&\quad \wedge\ \texttt{(ok-inputs x y)}\\
&\quad \wedge\ s_f = \texttt{(run (sched x y) } s_i\texttt{))}\\
&\rightarrow\\
&(\ \texttt{(next-inst } s_f\texttt{)} = \texttt{'(HALT)}\\
&\quad \wedge\\
&\quad \texttt{(top (stack } s_f\texttt{))} = \texttt{(* x y))}.
\end{aligned}$$

Total Correctness of *g-program*

```
(   si = (make-state 0
                     (list x y)
                     nil
                     *g-program*)
  ∧ (ok-inputs x y)
  ∧ sf = (run (sched x y) si))
→
( (next-inst sf) = '(HALT)
  ∧
   (top (stack sf)) = (* x y)).
```

Total Correctness of `*g-program*`

```
(   s_i = (make-state 0
                      (list x y)
                      nil
                      *g-program*)
  ∧ (natp x) ∧ (natp y)
  ∧ s_f = (run (sched x y) s_i))
→
( (next-inst s_f) = '(HALT)
  ∧
  (top (stack s_f)) = (* x y)).
```

Total Correctness of *g-program*

$$
\begin{aligned}
(\quad s_i \; = \; & \texttt{(make-state 0} \\
& \quad \texttt{(list x y)} \\
& \quad \texttt{nil} \\
& \quad \texttt{*g-program*)} \\
\wedge \; & \texttt{(natp x)} \wedge \texttt{(natp y)} \\
\wedge \; & s_f = \texttt{(run (sched x y) } s_i\texttt{))} \\
\rightarrow & \\
(\; & \texttt{(next-inst } s_f\texttt{)} = \texttt{'(HALT)} \\
\wedge & \\
& \texttt{(top (stack } s_f\texttt{))} = \texttt{(* x y)).}
\end{aligned}
$$

How Long Does *g-program* Run?

```
(defconst *g-program*
        '((ICONST 0)    ; 0
          (ISTORE 2)    ; 1  a = 0;
          (ILOAD 0)     ; 2  [loop:]
          (IFEQ 10)     ; 3  if x=0 then go to end;
          (ILOAD 0)     ; 4
          (ICONST 1)    ; 5
          (ISUB)        ; 6
          (ISTORE 0)    ; 7  x = x-1;
          (ILOAD 1)     ; 8
          (ILOAD 2)     ; 9
          (IADD)        ;10
          (ISTORE 2)    ;11  a = y+a;
          (GOTO -10)    ;12  go to loop
          (ILOAD 2)     ;13  [end:]
          (HALT)))      ;14 ``return'' a
```

PC at Loop and X=0

```
(defconst *g-program*
        '((ICONST 0)    ; 0
          (ISTORE 2)    ; 1   a = 0;
          (ILOAD 0)     ; 2   [loop:]
          (IFEQ 10)     ; 3   if x=0 then go to end;
          (ILOAD 0)     ; 4
          (ICONST 1)    ; 5
          (ISUB)        ; 6
          (ISTORE 0)    ; 7   x = x-1;
          (ILOAD 1)     ; 8
          (ILOAD 2)     ; 9
          (IADD)        ;10
          (ISTORE 2)    ;11   a = y+a;
          (GOTO -10)    ;12   go to loop
          (ILOAD 2)     ;13   [end:]
          (HALT)))      ;14 ``return'' a
```

PC at Loop and X>0

```
(defconst *g-program*
       '((ICONST 0)    ; 0
         (ISTORE 2)    ; 1   a = 0;
         (ILOAD 0)     ; 2   [loop:]
         (IFEQ 10)     ; 3   if x=0 then go to end;
         (ILOAD 0)     ; 4
         (ICONST 1)    ; 5
         (ISUB)        ; 6
         (ISTORE 0)    ; 7   x = x-1;
         (ILOAD 1)     ; 8
         (ILOAD 2)     ; 9
         (IADD)        ;10
         (ISTORE 2)    ;11   a = y+a;
         (GOTO -10)    ;12   go to loop
         (ILOAD 2)     ;13   [end:]
         (HALT)))      ;14 ``return'' a
```

PC at Top

```
(defconst *g-program*
       '((ICONST 0)    ; 0
         (ISTORE 2)    ; 1   a = 0;
         (ILOAD 0)     ; 2   [loop:]
         (IFEQ 10)     ; 3   if x=0 then go to end;
         (ILOAD 0)     ; 4
         (ICONST 1)    ; 5
         (ISUB)        ; 6
         (ISTORE 0)    ; 7   x = x-1;
         (ILOAD 1)     ; 8
         (ILOAD 2)     ; 9
         (IADD)        ;10
         (ISTORE 2)    ;11   a = y+a;
         (GOTO -10)    ;12   go to loop
         (ILOAD 2)     ;13   [end:]
         (HALT)))      ;14 ``return'' a
```

The Schedule for *g-program*

```
(defun g-loop-sched (x)
  (if (zp x)
      (repeat 'tick 3)
      (ap (repeat 'tick 11)
          (g-loop-sched (- x 1)))))

(defun g-sched (x)
  (ap (repeat 'tick 2)
      (g-loop-sched x)))
```

Demo 2

Total Correctness of *g-program*

$$
\begin{array}{l}
(\quad s_i = \texttt{(make-state 0} \\
\qquad\qquad\qquad\qquad \texttt{(list x y)} \\
\qquad\qquad\qquad\qquad \texttt{nil} \\
\qquad\qquad\qquad\qquad \texttt{*g-program*)} \\
\quad \wedge \texttt{(natp x)} \wedge \texttt{(natp y)} \\
\quad \wedge\ s_f = \texttt{(run (sched x y)}\ s_i\texttt{))} \\
\rightarrow \\
(\texttt{(next-inst}\ s_f\texttt{)} = \texttt{'(HALT)} \\
\quad \wedge \\
\quad \texttt{(top (stack}\ s_f\texttt{))} = \texttt{(* x y)).}
\end{array}
$$

Total Correctness of *g-program*

```
(   s_i = (make-state 0
                      (list x y)
                      nil
                      *g-program*)
  ∧ (natp x) ∧ (natp y)
  ∧ s_f = (run (g-sched x) s_i))
→
( (next-inst s_f) = '(HALT)
  ∧
   (top (stack s_f)) = (* x y)).
```


MAMOD

Moving Parts of M1 Proof Engine

- schedule functions $\surd$
- symbolic evaluation
- sequential composition
- two-step approach
- strong form of code correctness
- inner-loops first

Symbolic Evaluation

Suppose natp: `x,y,a` and ¬`(zp x)`.

```
(run (repeat 'tick 11)

     (make-state 2
                 (list x y a)
                 nil
                 *g-program*))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and ¬`(zp x)`.

```
(run '(tick ...10)

      (make-state 2
                  (list x y a)
                  nil
                  *g-program*))
```

Symbolic Evaluation

```
(defun run (sched s)
  (if (endp sched)
      s
      (run (cdr sched) (step s))))
```

Symbolic Evaluation

Suppose natp: x,y,a and $\neg$(zp x).

```
(run '(tick ...10)

      (make-state 2
                  (list x y a)
                  nil
                  *g-program*))
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(run '(tick ...9)
      (step
       (make-state 2
                   (list x y a)
                   nil
                   *g-program*)))
```

Symbolic Evaluation

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

(defun do-inst (inst s)
  (if (equal (op-code inst) 'ILOAD)
      (execute-ILOAD  inst s)
      (if (equal (op-code inst) 'ICONST)
          (execute-ICONST  inst s)
          ...)))
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(run '(tick ...9)
     (step
      (make-state 2
                  (list x y a)
                  nil
                  *g-program*)))
```

Symbolic Evaluation

Suppose natp: x,y,a and $\neg$(zp x).

```
(run '(tick ...9)
     (execute-ILOAD '(ILOAD 0)
      (make-state 2
                  (list x y a)
                  nil
                  *g-program*)))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(run '(tick ...9)

     (make-state 3
                 (list x y a)
                 (push x nil)
                 *g-program*))
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(run '(tick ...8)
     (execute-IFLE '(IFLE 10)
      (make-state 3
                  (list x y a)
                  (push x nil)
                  *g-program*)))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(run '(tick ...8)

      (make-state 4
                  (list x y a)
                  nil
                  *g-program*))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(run '(tick ...7)
     (execute-ILOAD '(ILOAD 0)
      (make-state 4
                  (list x y a)
                  nil
                  *g-program*)))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(run '(tick ...6)
     (execute-ICONST '(ICONST 1)
      (make-state 5
                  (list x y a)
                  (push x nil)
                  *g-program*)))
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(run '(tick ...5)
      (execute-ISUB '(ISUB)
       (make-state 6
                   (list x y a)
                   (push 1 (push x nil))
                   *g-program*)))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(run '(tick ...4)
     (execute-ISTORE '(ISTORE 0)
      (make-state 7
                  (list x y a)
                  (push (- x 1) nil)
                  *g-program*)))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and $\neg$`(zp x)`.

```
(run '(tick ...₃)
      (execute-ILOAD '(ILOAD 1)
       (make-state 8
                   (list (- x 1) y a)
                   nil
                   *g-program*)))
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(run '(tick tick tick)
      (execute-ILOAD '(ILOAD 2)
       (make-state 9
                   (list (- x 1) y a)
                   (push y nil)
                   *g-program*)))
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(run '(tick tick)
     (execute-IADD '(IADD)
      (make-state 10
                  (list (- x 1) y a)
                  (push a (push y nil))
                  *g-program*)))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and ¬`(zp x)`.

```
(run '(tick)
     (execute-ISTORE '(ISTORE 2)
      (make-state 11
                  (list (- x 1) y a)
                  (push (+ y a) nil)
                  *g-program*)))
```

Symbolic Evaluation

Suppose natp: x,y,a and ¬(zp x).

```
(run '()
      (execute-GOTO '(GOTO -10)
       (make-state 12
                   (list (- x 1) y (+ y a))
                   nil
                   *g-program*)))
```

Symbolic Evaluation

Suppose natp: `x`,`y`,`a` and ¬`(zp x)`.

```
(run '()

     (make-state 2
                 (list (- x 1) y (+ y a))
                 nil
                 *g-program*))
```

Symbolic Evaluation

Suppose natp: x,y,a and $\neg$(zp x).

```
(make-state 2
            (list (- x 1) y (+ y a))
            nil
            *g-program*)
```

Symbolic Evaluation

Thm.

```
(implies (and (natp x) (natp y) (natp a)
              (not (zp x)))
         (equal (run (repeat 'tick 11)
                     (make-state 2
                                 (list x y a)
                                 nil
                                 *g-program*))
                (make-state 2
                            (list (- x 1) y (+ y a))
                            nil
                            *g-program*)))
```

Demo 3

Demo 3

So if you have a formal specification of a programming language you can use a theorem prover as a symbolic evaluator!

Moving Parts of M1 Proof Engine

- schedule functions $\surd$
- symbolic evaluation $\surd$
- sequential composition
- two-step approach
- strong form of code correctness
- inner-loops first

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. `By induction on a.`

Aside on Induction

Thm. $\forall$ a,b,s : (ϕ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow$ $\forall$ b,s : (ϕ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$

$\forall$ b,s : (ϕ (cdr a) b s))

$\rightarrow$

$\forall$ b,s : (ϕ a b s).

Aside on Induction

Thm. $\forall$ a,b,s : (ϕ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow$ $\forall$ b,s : (ϕ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$

$\forall$ b,s : (ϕ (cdr a) b s))

$\rightarrow$

$\forall$ b,s : (ϕ a b s).

Aside on Induction

Thm. $(\phi$ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow \forall$ b,s : $(\phi$ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$

$\forall$ b,s : $(\phi$ (cdr a) b s))

$\rightarrow$

$\forall$ b,s : $(\phi$ a b s).

Aside on Induction

Thm. (ϕ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow$ $\forall$ b,s : (ϕ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$

$\forall$ b,s : (ϕ (cdr a) b s))

$\rightarrow$

$\forall$ b,s : (ϕ a b s).

Aside on Induction

Thm. (ϕ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow$ (ϕ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$

$\forall$ b,s : (ϕ (cdr a) b s))

$\rightarrow$

$\forall$ b,s : (ϕ a b s).

Aside on Induction

Thm. $(\phi$ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow$ $(\phi$ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$

$\forall$ b,s : $(\phi$ (cdr a) b s))

$\rightarrow$

$\forall$ b,s : $(\phi$ a b s).

Aside on Induction

Thm. `(`ϕ `a b s)`

Proof. `By induction on a.`

Base:

`(endp a)` $\rightarrow$ `(`ϕ `a b s)`

Induction Step:

`(` $\neg$`(endp a)`

$\wedge$

$\forall$ `b,s : (`ϕ `(cdr a) b s))`

$\rightarrow$

`(`ϕ `a b s).`

Aside on Induction

Thm. (ϕ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow$ (ϕ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$

$\forall$ b,s : (ϕ (cdr a) b s))

$\rightarrow$

(ϕ a b s).

Aside on Induction

Thm. $(\phi$ `a b s`$)$

Proof. `By induction on a.`

Base:

`(endp a)` $\rightarrow$ $(\phi$ `a b s`$)$

Induction Step:

`( ¬(endp a)`

$\wedge$ $(\phi$ `(cdr a)` β_1 $\delta_1)$

$\wedge$ $\forall$ `b,s :` $(\phi$ `(cdr a) b s))`

$\rightarrow$

$(\phi$ `a b s).`

Aside on Induction

Thm. $(\phi$ `a b s`$)$

Proof. `By induction on a.`

Base:

(endp a) $\rightarrow$ (ϕ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$ (ϕ (cdr a) β_1 δ_1) $\wedge$ (ϕ (cdr a) β_2 δ_2)

$\wedge$ $\forall$ b,s : (ϕ (cdr a) b s))

$\rightarrow$

(ϕ a b s).

Aside on Induction

Thm. (ϕ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow$ (ϕ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$ (ϕ (cdr a) β_1 δ_1) ... $\wedge$ (ϕ (cdr a) β_k δ_k)

$\wedge$ $\forall$ b,s : (ϕ (cdr a) b s))

$\rightarrow$

(ϕ a b s).

Aside on Induction

Thm. (ϕ a b s)

Proof. By induction on a.

Base:

(endp a) $\rightarrow$ (ϕ a b s)

Induction Step:

($\neg$(endp a)

$\wedge$ (ϕ (cdr a) β_1 δ_1) ... $\wedge$ (ϕ (cdr a) β_k δ_k))

$\rightarrow$

(ϕ a b s).

Sequential Composition

Thm. `(run (ap a b) s)` = `(run b (run a s))`

Proof. By induction on a.

Base:

(endp a)

$\rightarrow$

(run (ap a b) s) = (run b (run a s))

Sequential Composition

Thm. `(run (ap a b) s)` $=$ `(run b (run a s))`

Proof. `By induction on a.`

Base:

`(endp a)`

$\rightarrow$

`(run (ap a b) s)` $=$ `(run b (run a s))`

Sequential Composition

Thm. `(run (ap a b) s)` $=$ `(run b (run a s))`

Proof. `By induction on a.`

Base:

`(endp a)`

$\rightarrow$

`(run b s)` $=$ `(run b (run a s))`

Sequential Composition

Thm. `(run (ap a b) s)` $=$ `(run b (run a s))`

Proof. By induction on a.

Base:

(endp a)

$\rightarrow$

(run b s) $=$ (run b (run a s))

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. `By induction on a.`

Base:

`(endp a)`

$\rightarrow$

`(run b s) = (run b s)`

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. By induction on `a`.

Base:

`(endp a)`

$\rightarrow$

`(run b s) = (run b s)` $\surd$

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. By induction on a.

Base: √

Induction Step:

$$
\begin{array}{l}
(\ \neg\texttt{(endp a)} \\
\quad \wedge \\
\quad\quad \texttt{(run (ap (cdr a)}\ \beta\texttt{)}\ \delta\texttt{)} \\
\quad\quad\quad = \texttt{(run}\ \beta\ \texttt{(run (cdr a)}\ \delta\texttt{)))} \\
\to \\
\ \texttt{(run (ap a b) s)} \\
\quad\quad = \texttt{(run b (run a s))}
\end{array}
$$

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. By induction on a.

Base: $\surd$

Induction Step:

$$
\begin{array}{l}
(\ \neg(\texttt{endp a}) \\
\quad \wedge \\
\quad\quad (\texttt{run (ap (cdr a)}\ \beta)\ \delta) \\
\quad\quad\quad = (\texttt{run}\ \beta\ (\texttt{run (cdr a)}\ \delta))) \\
\rightarrow \\
\quad (\texttt{run (ap a b) s}) \\
\quad\quad = (\texttt{run b (run a s)})
\end{array}
$$

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. `By induction on a.`

Base: $\surd$

Induction Step:

$$
\begin{array}{l}
(\ \neg\texttt{(endp a)} \\
\quad \wedge \\
\quad\quad \texttt{(run (ap (cdr a) } \beta\texttt{) } \delta\texttt{)} \\
\quad\quad\quad = \texttt{(run } \beta \texttt{ (run (cdr a) } \delta\texttt{)))} \\
\rightarrow \\
\quad \texttt{(run (cons (car a) (ap (cdr a) b)) s)} \\
\quad\quad = \texttt{(run b (run a s))}
\end{array}
$$

Sequential Composition

Thm. `(run (ap a b) s)` $=$ `(run b (run a s))`

Proof. By induction on a.

Base: $\surd$

Induction Step:

($\neg$(endp a)

$\wedge$

(run (ap (cdr a) β) δ)

$=$ (run β (run (cdr a) δ)))

$\rightarrow$

(run (cons (car a) (ap (cdr a) b)) s)

$=$ (run b (run a s))

Sequential Composition

Thm. (run (ap a b) s) = (run b (run a s))

Proof. By induction on a.

Base: $\surd$

Induction Step:

($\neg$(endp a)

$\wedge$

(run (ap (cdr a) β) δ)

= (run β (run (cdr a) δ)))

$\rightarrow$

(run (ap (cdr a) b) (step s))

= (run b (run a s))

Sequential Composition

Thm. `(run (ap a b) s)` = `(run b (run a s))`

Proof. By induction on a.

Base: $\surd$

Induction Step:

($\neg$(endp a)

$\wedge$

(run (ap (cdr a) β) δ)

= (run β (run (cdr a) δ)))

$\rightarrow$

(run (ap (cdr a) b) (step s))

= (run b (run a s))

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. By induction on a.

Base: $\surd$

Induction Step:

$$
\begin{array}{l}
(\ \neg\texttt{(endp a)} \\
\quad \wedge \\
\quad\quad \texttt{(run (ap (cdr a)}\ \beta\texttt{)}\ \delta\texttt{)} \\
\quad\quad\quad = \texttt{(run}\ \beta\ \texttt{(run (cdr a)}\ \delta\texttt{)))} \\
\rightarrow \\
\quad \texttt{(run (ap (cdr a) b) (step s))} \\
\quad\quad = \texttt{(run b (run (cdr a) (step s)))}
\end{array}
$$

Sequential Composition

Thm. `(run (ap a b) s)` $=$ `(run b (run a s))`

Proof. By induction on a.

Base: $\surd$

Induction Step:

($\neg$(endp a)

$\wedge$

(run (ap (cdr a) β) δ)

$=$ (run β (run (cdr a) δ)))

$\rightarrow$

(run (ap (cdr a) b) (step s))

$=$ (run b (run (cdr a) (step s)))

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. By induction on `a`.

Base: $\surd$

Induction Step:

```
( ¬(endp a)
  ∧
   (run (ap (cdr a) b) (step s))
    = (run b (run (cdr a) (step s))))
→
 (run (ap (cdr a) b) (step s))
   = (run b (run (cdr a) (step s)))
```

Sequential Composition

Thm. (run (ap a b) s) $=$ (run b (run a s))

Proof. By induction on a.

Base: $\surd$

Induction Step:

($\neg$(endp a)

$\wedge$

(run (ap (cdr a) b) (step s))

$=$ (run b (run (cdr a) (step s))))

$\rightarrow$

(run (ap (cdr a) b) (step s))

$=$ (run b (run (cdr a) (step s)))$\surd$

Sequential Composition

Thm. `(run (ap a b) s) = (run b (run a s))`

Proof. `By induction on a.`

Base: √

Induction Step: √

`Q.E.D.`

Moving Parts of M1 Proof Engine

- schedule functions $\surd$
- symbolic evaluation $\surd$
- sequential composition $\surd$
- two-step approach
- strong form of code correctness
- inner-loops first

Two-Step Approach

To prove total correctness:

$$
\begin{array}{l}
(\quad s_i = \texttt{(make-state 0} \\
\qquad\qquad\qquad\qquad \texttt{(list } v_1 \ldots \texttt{)} \\
\qquad\qquad\qquad\qquad \texttt{nil} \\
\qquad\qquad\qquad\qquad \pi \texttt{)} \\
\quad \wedge\ \texttt{(ok-inputs } v_1 \ldots \texttt{)} \\
\quad \wedge\ s_f = \texttt{(run (sched } v_1 \ldots \texttt{) } s_i \texttt{))} \\
\rightarrow \\
(\ \texttt{(next-inst } s_f \texttt{)} = \texttt{'(HALT)} \\
\quad \wedge \\
\qquad \texttt{(top (stack } s_f \texttt{))} = \theta \texttt{).}
\end{array}
$$

We will always decompose the proof into two big stages:

- the code π implements some algorithm $(fn\ \ v_1 \ldots)$
- the algorithm $(fn\ \ v_1 \ldots)$ satisfies (or is equal to) θ

In the case of `*g-program*`:

- *code* π: `*g-program*`
- *algorithm* $(fn\ \ v_1 \ldots)$: `(g x y 0)`
- *spec* θ: `(* x y)`

Moving Parts of M1 Proof Engine

- schedule functions $\surd$
- symbolic evaluation $\surd$
- sequential composition $\surd$
- two-step approach $\surd$
- strong form of code correctness
- inner-loops first

Code Correctness – Step 1

Instead of

```
(   si = (make-state 0
                     (list v1 ...)
                     nil
                     π)
  ∧ (ok-inputs v1 ...)
  ∧ sf = (run (sched v1 ...) si))
→
( (next-inst sf) = '(HALT)
  ∧
   (top (stack sf)) = θ).
```

Code Correctness – Step 1

We'll attack:

(ok-inputs $v_1 \ldots$)

$\rightarrow$

(run (sched $v_1 \ldots$)
(make-state 0 (list $v_1 \ldots$) nil π))

$=$

(make-state
??? ; *final pc pointing to* (HALT)
??? ; *final locals via* fn
??? ; *final stack via* fn
π)

g-program Correctness – Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  ??? ; final pc pointing to (HALT)
  ??? ; final locals via fn
  ??? ; final stack via fn
  *g-program*)
```

g-program Correctness – Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14                       ; final pc
  ???                      ; final locals via fn
  ???                      ; final stack via fn
  *g-program*)
```

g-program Correctness – Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14                        ; final pc
  (list 0 y (* x y))        ; final locals via fn
  ???                       ; final stack via fn
  *g-program*)
```

g-program Correctness – Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14                   ; final pc
  (list 0 y (g x y 0)) ; final locals via fn
  ???                  ; final stack via fn
  *g-program*)
```

g-program Correctness – Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14                   ; final pc
  (list 0 y (g x y 0)) ; final locals via fn
  (push (g x y 0) nil) ; final stack via fn
  *g-program*)
```

`*g-program*` Correctness – Step 2

```
((natp x) ∧ (natp y))
→
 (g x y 0) = (* x y)
```

Putting It All Together

```
(   s_i = (make-state 0
                      (list x y)
                      nil
                      *g-program*)
  ∧ (natp x) ∧ (natp y)
  ∧ s_f = (run (g-sched x) s_i))
→
( (next-inst s_f) = '(HALT)
  ∧
   (top (stack s_f)) = (* x y)).
```

Putting It All Together

$$
\begin{aligned}
&(\quad s_i = \texttt{(make-state 0} \\
&\qquad\qquad\qquad\qquad \texttt{(list x y)} \\
&\qquad\qquad\qquad\qquad \texttt{nil} \\
&\qquad\qquad\qquad\qquad \texttt{*g-program*)} \\
&\ \wedge\ \texttt{(natp x)} \wedge \texttt{(natp y)} \\
&\ \wedge\ s_f = \texttt{(run (g-sched x)}\ s_i\texttt{))} \\
&\rightarrow \\
&(\ \texttt{(next-inst}\ s_f\texttt{)} = \texttt{'(HALT)} \\
&\ \wedge \\
&\quad \texttt{(top (stack}\ s_f\texttt{))} = \texttt{(* x y)).}
\end{aligned}
$$

Putting It All Together

((natp x) $\wedge$ (natp y)

$\wedge$ s_f $=$ (run (g-sched x)

(make-state 0

(list x y)

nil

g-program))

$\rightarrow$

((next-inst s_f) $=$ '(HALT)

$\wedge$

(top (stack s_f)) $=$ (* x y)).

Putting It All Together

```
((natp x) ∧ (natp y)
 ∧ s_f = (run (g-sched x)
              (make-state 0
                          (list x y)
                          nil
                          *g-program*))
→
( (next-inst s_f) = '(HALT)
 ∧
  (top (stack s_f)) = (* x y)).
```

Putting It All Together

((natp x) $\wedge$ (natp y)

$\wedge$ s_f = (run (g-sched x)
(make-state 0
(list x y)
nil
g-program))

$\rightarrow$

((next-inst s_f) = '(HALT)

$\wedge$

(top (stack s_f)) = (* x y)).

Putting It All Together

$$((\texttt{natp x}) \wedge (\texttt{natp y}) \wedge s_f = (\texttt{make-state}\ 14\ (\texttt{list 0 y (g x y 0)})\ (\texttt{push (g x y 0) nil})\ \texttt{*g-program*})$$

$$\rightarrow$$

$$((\texttt{next-inst}\ s_f) = \texttt{'(HALT)} \wedge (\texttt{top (stack}\ s_f)) = (\texttt{* x y})).$$

Putting It All Together

```
((natp x) ∧ (natp y)
 ∧ s_f = (make-state
                 14
                 (list 0 y (g x y 0))
                 (push (g x y 0) nil)
                 *g-program*)
→
( (next-inst s_f) = '(HALT)
 ∧
  (top (stack s_f)) = (* x y)).
```

$$((\texttt{natp x}) \land (\texttt{natp y}) \land s_f = (\texttt{make-state } 14 \; (\texttt{list 0 y (g x y 0)}) \; (\texttt{push (g x y 0) nil}) \; \texttt{*g-program*})) \rightarrow ((\texttt{next-inst } s_f) = \texttt{'(HALT)} \land (\texttt{top (stack } s_f\texttt{))} = (\texttt{* x y})).$$

Putting It All Together

```
((natp x) ∧ (natp y)
 ∧ s_f = (make-state
            14
            (list 0 y (* x y))
            (push (* x y) nil)
            *g-program*)
→
( (next-inst s_f) = '(HALT)
 ∧
  (top (stack s_f)) = (* x y)).
```

Putting It All Together

((natp x) $\wedge$ (natp y)

 $\wedge$ s_f = (make-state

                14

                (list 0 y (* x y))

                (push (* x y) nil)

                g-program)

$\rightarrow$

((next-inst s_f) = '(HALT)

 $\wedge$

  (top (stack s_f)) = (* x y)).

Putting It All Together

$((\texttt{natp x}) \land (\texttt{natp y})$

$\land\ s_f = \texttt{(make-state}$

$\qquad\qquad\texttt{14}$

$\qquad\qquad\texttt{(list 0 y (* x y))}$

$\qquad\qquad\texttt{(push (* x y) nil)}$

$\qquad\qquad\texttt{*g-program*)}$

$\rightarrow$

$(\ (\texttt{next-inst}\ s_f) = \texttt{'(HALT)}$

$\land$

$\quad(\texttt{top (stack}\ s_f\texttt{))} = \texttt{(* x y))}.$

Q.E.D.

Moving Parts of M1 Proof Engine

- schedule functions $\surd$
- symbolic evaluation $\surd$
- sequential composition $\surd$
- two-step approach $\surd$
- strong form of code correctness $\surd$
- inner-loops first

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Specify and prove the inner loop first!

Inner-Loop

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Inner-Loop

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Inner-Loop

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Inner-Loop

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Inner-Loop

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Inner-Loop

$((\texttt{natp x}) \wedge (\texttt{natp y}))$

$\rightarrow$

```
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Inner-Loop

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Inner-Loop

$((\texttt{natp x}) \wedge (\texttt{natp y}) \wedge (\texttt{natp a}))$

$\rightarrow$

```
(run (g-loop-sched x)
     (make-state 2 (list x y a) nil *g-progra
```

$=$

```
(make-state
 14
 (list 0 y (g x y 0))
 (push (g x y 0) nil)
 *g-program*)
```

Inner-Loop

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-program
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Inner-Loop

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Proof: Induct on `x`. *Base*: `(zp x)`

$((\texttt{natp x}) \wedge (\texttt{natp y}) \wedge (\texttt{natp a}))$

$\rightarrow$

```
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Proof: Induct on x. *Base*: (zp x)

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Proof: Induct on `x`. *Base*: `(zp x)`

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (repeat 'tick 3)
      (make-state 2 (list x y a) nil *g-program
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Proof: Induct on `x`. *Base*: `(zp x)`

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (repeat 'tick 3)
      (make-state 2 (list x y a) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Trivial by symbolic evaluation

Proof: Induct on `x`. *Step*: ¬`(zp x)`

Induction Hyp:

`((natp (- x 1))` ∧ `(natp` β`)` ∧ `(natp` α`))`

→

```
(run (g-loop-sched (- x 1))
     (make-state 2 (list (- x 1) β α) nil *g-
```

=

```
(make-state
 14
 (list 0 β (g (- x 1) β α))
 (push (g (- x 1) β α) nil)
 *g-program*)
```

Proof: Induct on `x`. *Step*: $\neg$`(zp x)`

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-program
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Proof: Induct on `x`. *Step*: $\neg$`(zp x)`

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-program
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Proof: Induct on x. *Step*: $\neg$(zp x)

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (ap (repeat 'tick 11)
          (g-loop-sched (- x 1)))
      (make-state 2 (list x y a) nil *g-progra
 = (make-state
    14
    (list 0 y (g x y a))
    (push (g x y a) nil)
    *g-program*)
```

Proof: Induct on x. *Step*: ¬(zp x)

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (ap (repeat 'tick 11)
          (g-loop-sched (- x 1)))
      (make-state 2 (list x y a) nil *g-progra
 = (make-state
    14
    (list 0 y (g x y a))
    (push (g x y a) nil)
    *g-program*)
```

Proof: Induct on x. *Step*: $\neg$(zp x)

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched (- x 1))
  (run (repeat 'tick 11)
       (make-state 2 (list x y a) nil *g-progr
 = (make-state
    14
    (list 0 y (g x y a))
    (push (g x y a) nil)
    *g-program*)
```

Proof: Induct on `x`. *Step*: $\neg$`(zp x)`

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched (- x 1))
  (run (repeat 'tick 11)
       (make-state 2 (list x y a) nil *g-progr
 = (make-state
    14
    (list 0 y (g x y a))
    (push (g x y a) nil)
    *g-program*)
```

Proof: Induct on `x`. *Step*: $\neg$`(zp x)`

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched (- x 1))
      (make-state 2 (list (- x 1) y (+ y a)) n
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Proof: Induct on x. *Step*: ¬(zp x)

Induction Hyp:

```
((natp (- x 1)) ∧ (natp β) ∧ (natp α))
→
 (run (g-loop-sched (- x 1))
      (make-state 2 (list (- x 1) β α) nil *g-
 =
 (make-state
  14
  (list 0 β (g (- x 1) β α))
  (push (g (- x 1) β α) nil)
  *g-program*)
```

Proof: Induct on `x`. *Step*: $\neg$`(zp x)`
Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched (- x 1))
      (make-state 2 (list (- x 1) y (+ y a)) n
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Proof: Induct on `x`. *Step*: $\neg$`(zp x)`

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *g-program*) =
 (make-state 14
             (list 0 y (g x y a))
             (push (g x y a) nil)
             *g-program*)
```

Proof: Induct on `x`. *Step*: $\neg$`(zp x)`

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *g-program*) =
 (make-state 14
             (list 0 y (g x y a))
             (push (g x y a) nil)
             *g-program*)
```

Proof: Induct on x. *Step*: ¬(zp x)

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *g-program*) =
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *g-program*)
```

Proof: Induct on `x`. *Step*: $\neg$`(zp x)`

Induction Conclusion:

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *g-program*) =
 (make-state 14
             (list 0 y (g (- x 1) y (+ y a)))
             (push (g (- x 1) y (+ y a)) nil)
             *g-program*) √
```

Thm: Inner-Loop Correct!

```
((natp x) ∧ (natp y) ∧ (natp a))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y a) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y a))
  (push (g x y a) nil)
  *g-program*)
```

Q.E.D.

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Specify and prove the inner loop first!

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-sched x)
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (ap (repeat 'tick 2)
          (g-loop-sched x))
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (ap (repeat 'tick 2)
          (g-loop-sched x))
      (make-state 0 (list x y) nil *g-program*
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (run (repeat 'tick 2)
           (make-state 0 (list x y) nil *g-pro
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (run (repeat 'tick 2)
           (make-state 0 (list x y) nil *g-pro
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y 0) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (run (g-loop-sched x)
      (make-state 2 (list x y 0) nil *g-progra
 =
 (make-state
  14
  (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (make-state 14 (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
 =
 (make-state 14 (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
```

Proving Step 1

```
((natp x) ∧ (natp y))
→
 (make-state 14 (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*)
 =
 (make-state 14 (list 0 y (g x y 0))
  (push (g x y 0) nil)
  *g-program*) √
```

Demo 4

Proving Step 2

```
(g x y a) = (if (zp x)
                a
                (g (- x 1)
                   y
                   (+ y a)))
```

Thm: `(g x y 0)=(* x y)`

Lemma: `(g x y a)=(+ a (* x y))`

To prove $\forall\ x, y, a\ :\ \phi(x, y, a)$ by induction:

Base Case:

(zp x) $\rightarrow \forall\ y, a\ :\ \phi(x, y, a)$

Induction Step:

($\neg$(zp x)

$\wedge$

$(\forall\ y, a\ :\ \phi(x-1, y, a)))$

$\rightarrow$

$\forall\ y, a\ :\ \phi(x, y, a)$

To prove $\forall\ x, y, a\ :\ \phi(x, y, a)$ by induction:

Base Case:

(zp x) $\rightarrow \forall\ y, a\ :\ \phi(x, y, a)$

Induction Step:

$(\neg$(zp x)

$\wedge$

$(\forall\ y, a\ :\ \phi(x-1, y, a)))$

$\rightarrow$

$\forall\ y, a\ :\ \phi(x, y, a)$

To prove $\forall \ x, y, a \ : \ \phi(x, y, a)$ by induction:

Base Case:

(zp x) $\rightarrow \phi(x, y, a)$

Induction Step:

($\neg$(zp x)

$\wedge$

$(\forall \ y, a \ : \ \phi(x-1, y, a))$)

$\rightarrow$

$\phi(x, y, a)$

To prove $\forall\ x, y, a\ :\ \phi(x, y, a)$ by induction:

Base Case:

(zp x) $\rightarrow\ \phi(x, y, a)$

Induction Step:

($\neg$(zp x)

$\wedge$

$(\forall\ y, a\ :\ \phi(x-1, y, a))$)

$\rightarrow$

$\phi(x, y, a)$

To prove $\forall\ x, y, a\ :\ \phi(x, y, a)$ by induction:

Base Case:

(zp x) $\rightarrow\ \phi(x, y, a)$

Induction Step:

($\neg$(zp x)

$\wedge$

$\phi(x-1, \gamma_1, \alpha_1)\ \wedge \ldots \wedge\ \phi(x-1, \gamma_k, \alpha_k)$)

$\rightarrow$

$\phi(x, y, a)$

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Base Case:

`(zp x)` $\rightarrow$ `(g x y a)=(+ a (* x y))`

Lemma: `(g x y a)`=`(+ a (* x y))`

Proof: `Induct on x.`

Base Case:

`(zp x)` $\rightarrow$ `(g x y a)`=`(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Base Case:

`(zp x)` $\rightarrow$ `a=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Base Case:

`(zp x)` $\rightarrow$ `a=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Base Case:

`(zp x)` $\rightarrow$ `a=(+ a 0)`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Base Case:

`(zp x)` $\rightarrow$ `a=(+ a 0)`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Base Case:

`(zp x)` $\rightarrow$ `a=a.`

Lemma: `(g x y a)`$=$`(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

`(`$\neg$`(zp x)`

$\wedge$

`(g (- x 1)` γ α`)`$=$`(+` α `(* (- x 1)` γ`)))`

$\rightarrow$

`(g x y a)`$=$`(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$(\neg$`(zp x)`

$\wedge$

`(g (- x 1)` γ α`)=(+` α `(* (- x 1)` γ`)))`

$\rightarrow$

`(g x y a)=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$(\neg$`(zp x)`

$\wedge$

`(g (- x 1)` γ α`)=(+` α `(* (- x 1)` γ`)))`

$\rightarrow$

`(g (- x 1) y (+ y a))=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$(\neg$`(zp x)`$)$

$\wedge$

`(g (- x 1)` γ α`)=(+` α `(* (- x 1)` γ`)))`

$\rightarrow$

`(g (- x 1) y (+ y a))=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$(\neg$`(zp x)`

$\wedge$

`(g (- x 1) y` α`)=(+` α `(* (- x 1) y)))`

$\rightarrow$

`(g (- x 1) y (+ y a))=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$$
\begin{array}{l}
\texttt{(}\neg\texttt{(zp x)} \\
\;\wedge \\
\;\texttt{(g (- x 1) y } \alpha\texttt{)=(+ } \alpha \texttt{ (* (- x 1) y)))} \\
\rightarrow \\
\;\texttt{(g (- x 1) y (+ y a))=(+ a (* x y))}
\end{array}
$$

Lemma: (g x y a)=(+ a (* x y))

Proof: Induct on x.

Induction Step

(¬(zp x)

$\wedge$

(g (- x 1) y (+ y a))=(+ (+ y a) (* (- x 1) y

$\rightarrow$

(g (- x 1) y (+ y a))=(+ a (* x y))

Lemma: (g x y a)=(+ a (* x y))

Proof: Induct on x.

Induction Step

(¬(zp x)

∧

(g (- x 1) y (+ y a))=(+ (+ y a) (* (- x 1) y

→

(g (- x 1) y (+ y a))=(+ a (* x y))

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$($¬`(zp x)`

$\wedge$

`(g (- x 1) y (+ y a))=(+ (+ y a) (* (- x 1) y`

$\rightarrow$

`(g (- x 1) y (+ y a))=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

`(`$\neg$`(zp x)`

$\wedge$

`(g (- x 1) y (+ y a))=(+ y a (* (- x 1) y)))`

$\rightarrow$

`(g (- x 1) y (+ y a))=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

(¬(zp x)

$\wedge$

(g (- x 1) y (+ y a))=(+ y a (* (- x 1) y)))

$\rightarrow$

(g (- x 1) y (+ y a))=(+ a (* x y))

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$(\neg$`(zp x)`

$\wedge$

`(g (- x 1) y (+ y a))=(+ a y (* (- x 1) y)))`

$\rightarrow$

`(g (- x 1) y (+ y a))=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$(\neg$`(zp x)`

$\wedge$

`(g (- x 1) y (+ y a))=(+ a y (* (- x 1) y)))`

$\rightarrow$

`(g (- x 1) y (+ y a))=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

(¬(zp x)

 ∧

 (g (- x 1) y (+ y a))=(+ a y (* x y) (- y)))

→

 (g (- x 1) y (+ y a))=(+ a (* x y))

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$(\neg$`(zp x)`

$\wedge$

`(g (- x 1) y (+ y a))=(+ a y (* x y) (- y)))`

$\rightarrow$

`(g (- x 1) y (+ y a))=(+ a (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

$(\neg$(zp x)

$\wedge$

(g (- x 1) y (+ y a))=(+ a (* x y)))

$\rightarrow$

(g (- x 1) y (+ y a))=(+ a (* x y))

Lemma: `(g x y a)=(+ a (* x y))`

Proof: `Induct on x.`

Induction Step

```
(¬(zp x)
 ∧
 (g (- x 1) y (+ y a))=(+ a (* x y)))
→
 (g (- x 1) y (+ y a))=(+ a (* x y))
```

Q.E.D.

Lemma: `(g x y a)=(+ a (* x y))`

Thm: `(g x y 0) = (* x y)`

Lemma: (g x y a)=(+ a (* x y))

Thm: (g x y 0) = (* x y)

Proof: (g x y 0) =

Lemma: `(g x y a)=(+ a (* x y))`

Thm: `(g x y 0) = (* x y)`

Proof: `(g x y 0) = (+ 0 (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Thm: `(g x y 0) = (* x y)`

Proof: `(g x y 0) = (+ 0 (* x y))`

Lemma: `(g x y a)=(+ a (* x y))`

Thm: `(g x y 0) = (* x y)`

Proof: `(g x y 0) = (+ 0 (* x y)) = (* x y)`

Lemma: `(g x y a)=(+ a (* x y))`

Thm: `(g x y 0) = (* x y)`

Proof: `(g x y 0) = (+ 0 (* x y)) = (* x y)`
`Q.E.D.`

Demo 5