```
#|

 Copyright (C) 1994 by Computational Logic, Inc.  All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Computational Logic, Inc. PROVIDES ABSOLUTELY NO WARRANTY.  THE EVENT SCRIPT
 IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,
 INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE DEFECTIVE, YOU
 ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

 IN NO EVENT WILL Computational Logic, Inc. BE LIABLE TO YOU FOR ANY DAMAGES,
 ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL
 DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT
 NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
 SUSTAINED BY THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

|#

; An NQTHM Formalization of a Small Machine

; by J Strother Moore II
; May 30, 1991

; This file serves as a good introduction to the Nqthm approach to
; language semantics.  We have carried out this approach on much larger
; examples than presented here.  It is, for example, that used at all
; levels of the CLI short stack (hardware description language, machine
; language, assembly language, high-level language).  The semantics of those
; levels are so large and complicated that it is difficult to see the
; basic ideas.  Those ideas are highlighted here simply by dealing with
; a trivial language.

; This is a list of events to be processed by NQTHM starting from the
; GROUND-ZERO state.  In it I develop
; (a) an operational semantics for a simple programming language
; (b) a program that implements multiplication by repeated addition
; (c) a proof of the correctness of the multiplication program
```

```
;     directly from the "operational" semantics
; (d) a program that does exponentiation and uses the multiplier
; (e) a proof of the correctness of the exponentiation program
; (f) the most general correctness theorem about the multiplier
; (g) the definition and correctness of the "McCarthy" functional
;     semantics of the multiplier
; (h) a proof of the correctness of the multiplier by the inductive
;     assertion method.
; (i) May 13, 1992.  a proof of the general theorem that our
;     standard form of a correctness result for a subroutine
;     implies our standard form of a termination result.  This
;     part of the file is not part of the tutorial because the
;     proof is pretty messy.

; The programming language is not particularly elegant.  Its only
; redeeming features are that its semantics is easily written down and
; it lets me illustrate the points I'm trying to make.  This is by no
; means a complete or exemplary "library" for dealing with programs in
; this language; I have in fact kept the facts to a bare minimum.

; We start by defining our "small machine."
```

EVENT: Start with the initial **nqthm** theory.

```
; States are represented by the following shell objects:
```

EVENT: Add the shell *st*, with recognizer function symbol *stp* and 5 accessors: *pc*, with type restriction (none-of) and default value zero; *stk*, with type restriction (none-of) and default value zero; *mem*, with type restriction (none-of) and default value zero; *haltedp*, with type restriction (none-of) and default value zero; *defs*, with type restriction (none-of) and default value zero.

```
; Utility Functions
```

DEFINITION: add1-pc$(pc) = \text{cons}(\text{car}(pc), 1 + \text{cdr}(pc))$

DEFINITION:
get(n, lst)
= **if** $n \simeq 0$ **then** car(lst)
else get$(n - 1, \text{cdr}(lst))$ **endif**

DEFINITION:
put $(n,\ v,\ lst)$
$= \quad$ **if** $n \simeq$ `0` **then** cons $(v,$ cdr $(lst))$
else cons (car (lst), put $(n - 1,\ v,$ cdr $(lst)))$ **endif**

DEFINITION:
fetch $(pc,\ defs) =$ get (cdr (pc), cdr (assoc (car (pc), $defs$)))

```
; The Semantics of Individual Instructions

; Move Instructions
```

DEFINITION:
move $(addr1,\ addr2,\ s)$
$= \quad$ st (add1-pc (pc (s)),
stk (s),
put $(addr1$, get $(addr2$, mem $(s))$, mem $(s))$,
f,
defs (s))

DEFINITION:
movi $(addr,\ val,\ s)$
$= \quad$ st (add1-pc (pc (s)), stk (s), put $(addr,\ val,$ mem $(s))$, **f**, defs (s))

```
; Arithmetic Instructions
```

DEFINITION:
add $(addr1,\ addr2,\ s)$
$= \quad$ st (add1-pc (pc (s)),
stk (s),
put $(addr1$, get $(addr1$, mem $(s)) +$ get $(addr2$, mem $(s))$, mem $(s))$,
f,
defs (s))

DEFINITION:
subi $(addr,\ val,\ s)$
$= \quad$ st (add1-pc (pc (s)),
stk (s),
put $(addr$, get $(addr$, mem $(s)) - val$, mem $(s))$,
f,
defs (s))

```
; Jump Instructions
```

Definition:
jumpz $(addr,\ pc,\ s)$
$=$ st (**if** get $(addr$, mem $(s)) \simeq$ `0` **then** cons (car (pc (s)), pc)
else add1-pc (pc (s)) **endif**,
stk (s),
mem (s),
f,
defs (s))

Definition:
jump $(pc,\ s)$ = st (cons (car (pc (s)), pc), stk (s), mem (s), **f**, defs (s))

```
; Subroutine Call and Return
```

Definition:
call $(subr,\ s)$
$=$ st (cons ($subr$, `0`), cons (add1-pc (pc (s)), stk (s)), mem (s), **f**, defs (s))

Definition:
ret (s)
$=$ **if** stk $(s) \simeq$ **nil** **then** st (pc (s), stk (s), mem (s), **t**, defs (s))
else st (car (stk (s)), cdr (stk (s)), mem (s), **f**, defs (s)) **endif**

```
; One can imagine adding new instructions.

; The Interpreter
```

Definition:
execute $(ins,\ s)$
$=$ **if** car (ins) = `'move` **then** move (cadr (ins), caddr (ins), s)
elseif car (ins) = `'movi` **then** movi (cadr (ins), caddr (ins), s)
elseif car (ins) = `'add` **then** add (cadr (ins), caddr (ins), s)
elseif car (ins) = `'subi` **then** subi (cadr (ins), caddr (ins), s)
elseif car (ins) = `'jumpz` **then** jumpz (cadr (ins), caddr (ins), s)
elseif car (ins) = `'jump` **then** jump (cadr (ins), s)
elseif car (ins) = `'call` **then** call (cadr (ins), s)
elseif car (ins) = `'ret` **then** ret (s)
else s **endif**

Definition:
step (s)
$=$ **if** haltedp (s) **then** s
else execute (fetch (pc (s), defs (s)), s) **endif**

DEFINITION:
sm $(s,\ n)$
$=$ **if** $n \simeq 0$ **then** s
else sm (step (s), $n - 1$) **endif**

```
; This concludes our formal definition of the intepreter.

; We next prove a small collection of lemmas that tightly control the
; expansion of the the interpreter.  The idea is that we don't want sm
; or step to expand unless we know what the current instruction is and
; have enough time on the clock to execute it.  So we will prove
; certain rewrite rules that manipulate step and sm and then disable
; those functions so that only the rules are available.
```

THEOREM: step-opener
(haltedp (s) $\rightarrow$ (step $(s) = s$))
$\wedge$ (listp (fetch (pc (s), defs (s))))
$\rightarrow$ (step (s)
$=$ **if** haltedp (s) **then** s
else execute (fetch (pc (s), defs (s)), s) **endif**)

EVENT: Disable step.

THEOREM: sm-plus
sm $(s,\ i + j)$ = sm (sm $(s,\ i)$, j)

THEOREM: sm-add1
sm $(s,\ 1 + i)$ = sm (step (s), i)

THEOREM: sm-0
sm $(s,\ 0) = s$

EVENT: Disable sm.

```
; Now we move to our first example program.  We will define a program
; that multiplies two naturals by successive addition.  We will then
; prove it correct.

; The program we have in mind is:

; (times (movi 2 0)
;        (jumpz 0 5)
;        (add 2 1)
```

```
;        (subi 0 1)
;        (jump 1)
;        (ret))

; Observe that the program multiplies the contents of reg 0 by the
; contents of reg 1 and leaves the result in reg 2.  At the end, reg 0
; is 0 and reg 1 is unchanged.  If we start at a (call times) this
; program requires 2+4i+2 instructions, where i is the initial
; contents of reg 0.  To keep the proof incredibly simple, we will
; prove the program correct only for the 5 register version of our
; machine!  (Why 5?  Why not 3?  Because eventually we will use times
; in another program that uses 5 registers.  In general we should
; prove it for an arbitrarily large memory -- and we will -- but that
; just complicates the statement without contributing to the example.)

; We start by defining the constant that is this program:
```

DEFINITION:
TIMES-PROGRAM
= '(times
      (movi 2 0)
      (jumpz 0 5)
      (add 2 1)
      (subi 0 1)
      (jump 1)
      (ret))

```
; 5  Return

; and a function that multiplies the "same way."
```

DEFINITION:
times-fn (i, j, ans)
$=$ **if** $i \simeq 0$ **then** ans
else times-fn $(i - 1, j, ans + j)$ **endif**

```
; In some sense, the following mathematical fact completely captures
; the correctness of the program:
```

THEOREM: times-fn-is-times
$(ans \in \mathbf{N}) \rightarrow (\text{times-fn}\,(i, j, ans) = ((i * j) + ans))$

```
; at least if one also understands
```

THEOREM: plus-right-id
$(x + 0) = \text{fix}(x)$

```
; The real problem is proving that the program has this semantics.
; First, how much time does the program need?  It takes one tick to do
; the CALL, one for the MOVI at pc 0, then 4 ticks for each iteration
; of the loop at pc 1, and then 2 more ticks to get out of the loop
; and do the RET.  So:
```

DEFINITION: $\text{times-clock}(i) = (2 + (i * 4) + 2)$

```
; We could have written (plus 4 (times i 4)) but by using this
; algebraically odd expression we make sm-plus, above, immediately
; applicable.

; We next address ourselves to the loop from pc 1 through 4.  Consider
; an arbitary arrival at pc 1 and suppose you have (times i 4) ticks.
; The following theorem tells us what you get:
```

THEOREM: times-correct-lemma
$$
\begin{array}{l}
((i \in \mathbf{N}) \wedge (\text{assoc}(\texttt{'times}, \textit{defs}) = \text{TIMES-PROGRAM})) \\
\rightarrow \quad (\text{sm}(\text{st}(\texttt{'(times . 1)}, \textit{stk1}, \text{list}(i, j, \textit{ans}, \textit{r3}, \textit{r4}), \mathbf{f}, \textit{defs}), i * 4) \\
\qquad = \quad \text{st}(\texttt{'(times . 1)}, \\
\qquad\qquad \textit{stk1}, \\
\qquad\qquad \text{list}(0, j, \text{times-fn}(i, j, \textit{ans}), \textit{r3}, \textit{r4}), \\
\qquad\qquad \mathbf{f}, \\
\qquad\qquad \textit{defs}))
\end{array}
$$

```
; It is then trivial to construct the entire correctness proof:
```

THEOREM: times-correct
$$
\begin{array}{l}
((\text{fetch}(\textit{pc}, \textit{defs}) = \texttt{'(call times)}) \\
\ \wedge \quad (\text{assoc}(\texttt{'times}, \textit{defs}) = \text{TIMES-PROGRAM}) \\
\ \wedge \quad (i \in \mathbf{N})) \\
\rightarrow \quad (\text{sm}(\text{st}(\textit{pc}, \textit{stk}, \text{list}(i, j, \textit{r2}, \textit{r3}, \textit{r4}), \mathbf{f}, \textit{defs}), \text{times-clock}(i)) \\
\qquad = \quad \text{st}(\text{add1-pc}(\textit{pc}), \textit{stk}, \text{list}(0, j, i * j, \textit{r3}, \textit{r4}), \mathbf{f}, \textit{defs}))
\end{array}
$$

```
; We disable the clock function so that subsequent programs can
; use it without its expansion messing up their algebraic
; form.
```

EVENT: Disable times-clock.

```
; It is worth noting that this file has been rather carefully crafted
; to make the above proof go through with a minimum of fuss.  In
; general, we will have to prove lots of lemmas about the utility
; functions GET and PUT to handle arbitrarily sized memories.  And we
; have to prove lots of lemmas about arithmetic to explain the data
; handling in our programs.  To some extent those arithmetic facts get
; in the way of our desired treatment of the clock in our proofs,
; e.g., if the theorem prover knows the usual facts about PLUS and
; TIMES then (PLUS 2 (TIMES I 4) 2) would become (PLUS 4 (TIMES 4 I))
; and we'd then have to take special care to force sm to open the way
; we want in this proof.  One avenue that has been used to avoid this
; problem is to define the clock functions with special arithmetic
; primitives, e.g., CLK-PLUS and CLK-TIMES (which are in fact just the
; familiar functions) but which we then disable and isolate from the
; free-wheeling arithmetic simplifications.

; We now consider the role of subroutine call and return in this
; language.  To illustrate it we'll implement exponentiation, which
; will CALL our TIMES program.  The proof of the correctness of the
; exponentiation program will rely on the correctness of TIMES, not on
; re-analysis of the code for TIMES.

; The mathematical function we wish to implement is:
```

DEFINITION:
$\exp(i, j)$
$= \textbf{if } j \simeq 0 \textbf{ then } 1$
$\quad \textbf{else } \exp(i, j - 1) * i \textbf{ endif}$

```
; The program we have in mind is:
```

DEFINITION:
EXP-PROGRAM
```
=  '(exp
     (move 3 0)
     (move 4 1)
     (movi 1 1)
     (jumpz 4 9)
     (move 0 3)
     (call times)
```

```
      (move 1 2)
      (subi 4 1)
      (jump 3)
      (ret))

; 9  return

; A recursive description of the loop (pc 3 through 8) in this
; algorithm is:
```

DEFINITION:
exp-fn $(r0, r1, r2, r3, r4)$
$=$ **if** $r4 \simeq 0$ **then** $r1$
else exp-fn $(0, r3 * r1, r3 * r1, r3, r4 - 1)$ **endif**

```
; Pretty weird.

; We need a little more arithmetic than we have, namely
; associativity and right identity for times:
```

THEOREM: associativity-of-times
$((i * j) * k) = (i * (j * k))$

THEOREM: times-right-id
$(i * 1) = \text{fix}(i)$

```
; So now the system can prove that the weird exp-fn is just exp (in a
; generalized sense that accomodates the initial value of r1).
```

THEOREM: exp-fn-is-exp
$(r1 \in \mathbf{N}) \rightarrow (\text{exp-fn}(r0, r1, r2, r3, r4) = (\exp(r3, r4) * r1))$

```
; Here is the clock function for exp.  Again we use an algebraically
; odd form simply to gain instant access to the desired sm-plus
; decomposition.  The "4" gets us past the CALL and the first 3
; initialization instructions; the times expression takes us around
; the exp loop j times, and the final "2" gets us out through the RET.
; Note that as we go around the loop we make explicit reference to
; TIMES-CLOCK to explain the CALL of TIMES.
```

DEFINITION:
exp-clock $(i, j) = (4 + (j * (2 + \text{times-clock}(i) + 3)) + 2)$

```
; Now we prove the "loop invariant" for the EXP program.  We simply
; tell the system to induct according to exp-fn.  We could "trick" it
; into doing that by using exp-fn in place of the (times (exp r3 r4)
; r1) expressions, but that is devious and doesn't always work.
```

THEOREM: exp-correct-lemma

$$
\begin{array}{l}
((r3 \in \mathbf{N}) \\
\;\wedge\;\; (r4 \in \mathbf{N}) \\
\;\wedge\;\; (\mathrm{assoc}\,(\texttt{'exp},\ \mathit{defs}) = \textsc{exp-program}) \\
\;\wedge\;\; (\mathrm{assoc}\,(\texttt{'times},\ \mathit{defs}) = \textsc{times-program})) \\
\rightarrow\;\; (\mathrm{sm}\,(\mathrm{st}\,(\texttt{'(exp . 3)},\ \mathit{stk},\ \mathrm{list}\,(r0,\ r1,\ r2,\ r3,\ r4),\ \mathbf{f},\ \mathit{defs}), \\
\qquad\quad r4 * (\texttt{2} + \text{times-clock}\,(r3) + \texttt{3})) \\
\qquad = \;\; \mathrm{st}\,(\texttt{'(exp . 3)}, \\
\qquad\qquad\quad \mathit{stk}, \\
\qquad\qquad\quad \mathbf{if}\ r4 \simeq \texttt{0}\ \ \mathbf{then}\ \mathrm{list}\,(r0,\ r1,\ r2,\ r3,\ r4) \\
\qquad\qquad\quad \mathbf{else}\ \mathrm{list}\,(\texttt{0}, \\
\qquad\qquad\qquad\qquad \exp\,(r3,\ r4) * r1, \\
\qquad\qquad\qquad\qquad \exp\,(r3,\ r4) * r1, \\
\qquad\qquad\qquad\qquad r3, \\
\qquad\qquad\qquad\qquad \texttt{0})\ \mathbf{endif}, \\
\qquad\qquad\quad \mathbf{f}, \\
\qquad\qquad\quad \mathit{defs}))
\end{array}
$$

```
; The theorem prover is now set up to prove that exp is correct
; without further assistance.  (But you must not underestimate how
; clever this assistance has been to make this possible!)
```

THEOREM: exp-correct

$$
\begin{array}{l}
((i \in \mathbf{N}) \\
\;\wedge\;\; (j \in \mathbf{N}) \\
\;\wedge\;\; (\mathrm{fetch}\,(\mathit{pc},\ \mathit{defs}) = \texttt{'(call exp)}) \\
\;\wedge\;\; (\mathrm{assoc}\,(\texttt{'exp},\ \mathit{defs}) = \textsc{exp-program}) \\
\;\wedge\;\; (\mathrm{assoc}\,(\texttt{'times},\ \mathit{defs}) = \textsc{times-program})) \\
\rightarrow\;\; (\mathrm{sm}\,(\mathrm{st}\,(\mathit{pc},\ \mathit{stk},\ \mathrm{list}\,(i,\ j,\ r2,\ r3,\ r4),\ \mathbf{f},\ \mathit{defs}),\ \text{exp-clock}\,(i,\ j)) \\
\qquad = \;\; \mathrm{st}\,(\text{add1-pc}\,(\mathit{pc}), \\
\qquad\qquad\quad \mathit{stk}, \\
\qquad\qquad\quad \mathbf{if}\ j \simeq \texttt{0}\ \ \mathbf{then}\ \mathrm{list}\,(i,\ \exp\,(i,\ j),\ r2,\ i,\ \texttt{0}) \\
\qquad\qquad\quad \mathbf{else}\ \mathrm{list}\,(\texttt{0},\ \exp\,(i,\ j),\ \exp\,(i,\ j),\ i,\ \texttt{0})\ \mathbf{endif}, \\
\qquad\qquad\quad \mathbf{f}, \\
\qquad\qquad\quad \mathit{defs}))
\end{array}
$$

```
; Ok, enough of this.  Presumably the point has been made:  correctness
; proofs can be "stacked."
```

```
; Recall that we have been dealing with an unnecessarily restricted
; view of the machine, namely that it only have 5 memory locations.
; Before leaving this approach and pursuing some others, let us
; quickly prove the most general form of the correctness result for
; TIMES.

; We start with the basic normalization rules for get and put.
```

DEFINITION:
length (lst)
$=$ **if** $lst \simeq$ **nil** **then** 0
else $1 +$ length (cdr (lst)) **endif**

THEOREM: put-put-0
$((addr < \text{length}(mem)) \land (\text{get}(addr,\ mem) = val))$
$\rightarrow\ (\text{put}(addr,\ val,\ mem) = mem)$

THEOREM: put-put-1
$\text{put}(addr,\ v2,\ \text{put}(addr,\ v1,\ mem)) = \text{put}(addr,\ v2,\ mem)$

THEOREM: put-put-2
$((addr1 \in \mathbf{N}) \land (addr2 \in \mathbf{N}) \land (addr1 \neq addr2))$
$\rightarrow\ (\text{put}(addr2,\ v2,\ \text{put}(addr1,\ v1,\ mem)) = \text{put}(addr1,\ v1,\ \text{put}(addr2,\ v2,\ mem)))$

THEOREM: get-put
$((addr1 \in \mathbf{N}) \land (addr2 \in \mathbf{N}))$
$\rightarrow$ (get ($addr1$, put ($addr2$, val, mem))
$=$ **if** $addr1 = addr2$ **then** val
else get ($addr1$, mem) **endif**)

THEOREM: length-put
$(addr < \text{length}(mem)) \rightarrow (\text{length}(\text{put}(addr,\ val,\ mem)) = \text{length}(mem))$

EVENT: Disable get.

EVENT: Disable put.

```
; And a few basic arithmetic facts.
```

THEOREM: difference-1
$(x - 1) = (x - 1)$

THEOREM: difference-elim

$((i \in \mathbf{N}) \wedge (i \not< j)) \rightarrow ((j + (i - j)) = i)$

THEOREM: associativity-of-plus

$((i + j) + k) = (i + (j + k))$

THEOREM: commutativity-of-plus

$(i + j) = (j + i)$

THEOREM: commutativity2-of-plus

$(i + (k + j)) = (k + (i + j))$

```
; Ok, now we get specific to the TIMES program.  The following function
; "is" loop in the TIMES program vis-a-vis its effect on a completely
; arbitrary memory mem.  If a program is run entirely for its effect on
; memory (as opposed to the subroutine stack or the haltedp flag, then
; this program "is" the McCarthy-esque functional analogue of the loop.
```

DEFINITION:

times-mem-fn-loop (mem)

$= $ **if** get $(0,\ mem) \simeq 0$ **then** mem

else times-mem-fn-loop (put $(0,$

get $(0,\ mem) - 1,$

put $(2,$ get $(2,\ mem) +$ get $(1,\ mem),\ mem)))$ **endif**

DEFINITION:

times-mem-fn $(mem) =$ times-mem-fn-loop (put $(2,\ 0,\ mem))$

```
; In proving this functional analogue correct we essentially carry
; our McCarthy's functional semantics approach.  The theorem below
; establishes that times-mem-fn-loop just does two puts into mem:  it
; 0's r0 and it puts (r0*r1)+r2 into location r2:
```

THEOREM: times-mem-fn-loop-is-times

$(($get $(0,\ mem) \in \mathbf{N}) \wedge ($get $(2,\ mem) \in \mathbf{N}) \wedge (2 <$ length $(mem)))$

$\rightarrow$ (times-mem-fn-loop (mem)

$=$ put $(0,$

$0,$

put $(2,$ (get $(0,\ mem) *$ get $(1,\ mem)) +$ get $(2,\ mem),\ mem)))$

THEOREM: times-mem-fn-is-correct

$(($get $(0,\ mem) \in \mathbf{N}) \wedge (2 <$ length $(mem)))$

$\rightarrow$ (times-mem-fn (mem)

$=$ put $(0,\ 0,$ put $(2,$ get $(0,\ mem) *$ get $(1,\ mem),\ mem)))$

```
; Our aim, in the revisited times-correct theorem, is to establish that
; executing a CALL of TIMES has the following effect on an almost arbitrary
; state s:
```

DEFINITION:
times-step (s)
$= \quad$ st (add1-pc (pc (s)),
stk (s),
put (0, 0, put (2, get (0, mem (s)) $*$ get (1, mem (s)), mem (s))),
f,
defs (s))

```
; The proof proceeds, as we have seen twice before, first by an
; inductive analysis of the loop itself.  Note that we induct
; according to times-mem-fn-loop.
```

THEOREM: times-correct-lemma-revisited
((get (0, *mem*) $\in$ **N**) $\wedge$ (assoc (`'times`, *defs*) = TIMES-PROGRAM))
$\rightarrow$ (sm (st (`'(times . 1)`, *stk1*, *mem*, **f**, *defs*), get (0, *mem*) $*$ 4)
= st (`'(times . 1)`, *stk1*, times-mem-fn-loop (*mem*), **f**, *defs*))

```
; Unfortunately, the above lemma is not quite applicable in our use below
; because the mem that occurs in the state in the lhs of the conclusion is
; not going to be syntactically identical to the mem that occurs in the
; (times (get 0 mem) 4) in the clock.  The reason is that the clock mem is
; the original mem while the state mem is the one produced by moving a 0
; into r2.  Of course, they have the same r0 value.  So, having proved
; the inductive fact we need, we now "generalize" it.
```

THEOREM: times-correct-lemma-revisited-and-generalized
((*r0* = get (0, *mem*))
$\wedge$ (get (0, *mem*) $\in$ **N**)
$\wedge$ (assoc (`'times`, *defs*) = TIMES-PROGRAM))
$\rightarrow$ (sm (st (`'(times . 1)`, *stk1*, *mem*, **f**, *defs*), *r0* $*$ 4)
= st (`'(times . 1)`, *stk1*, times-mem-fn-loop (*mem*), **f**, *defs*))

```
; And now we can prove the most general form of the correctness of our
; TIMES program.  It tells us that if you are interested in (sm s n),
; where the pc points to a CALL of TIMES, the definition of 'TIMES is
; ours, memory is at least 3 long, r0 is numeric, the halt flag is
; off, and there are at least (times-clock r0) ticks on the clock,
; then you can just take a times-step and decrease the clock by
; (times-clock r0).  What more could you want?
```

THEOREM: times-correct-revisited

$$
\begin{array}{ll}
((\text{fetch}(\text{pc}(s), \text{defs}(s)) = \texttt{'(call times)}) \\
\ \wedge & (\text{assoc}(\texttt{'times}, \text{defs}(s)) = \text{TIMES-PROGRAM}) \\
\ \wedge & (2 < \text{length}(\text{mem}(s))) \\
\ \wedge & (\mathit{r0} = \text{get}(0, \text{mem}(s))) \\
\ \wedge & (\mathit{r0} \in \mathbf{N}) \\
\ \wedge & (n \not< \text{times-clock}(\mathit{r0})) \\
\ \wedge & (\neg\ \text{haltedp}(s))) \\
\rightarrow & (\text{sm}(s,\ n) = \text{sm}(\text{times-step}(s),\ n - \text{times-clock}(\mathit{r0})))
\end{array}
$$

```
; The Inductive Assertion Approach

; First, we simply prove the hand-generated verification
; conditions from an informal annotation of our TIMES
; program.
```

THEOREM: verification-conditions-for-times

$$
\begin{array}{l}
(((\mathit{i0} \in \mathbf{N}) \wedge (\mathit{i1} \in \mathbf{N})) \\
\quad \rightarrow \quad ((0 \in \mathbf{N}) \wedge ((\mathit{i0} * \mathit{i1}) = (0 + (\mathit{i0} * \mathit{i1}))))) \\
\wedge \quad (((\mathit{r2} \in \mathbf{N}) \wedge ((\mathit{i0} * \mathit{i1}) = (\mathit{r2} + (\mathit{r0} * \mathit{r1}))) \wedge (\mathit{r0} \not\simeq 0)) \\
\qquad \rightarrow \quad (((\mathit{r2} + \mathit{r1}) \in \mathbf{N}) \\
\qquad\qquad \wedge \quad ((\mathit{i0} * \mathit{i1}) = ((\mathit{r2} + \mathit{r1}) + ((\mathit{r0} - 1) * \mathit{r1}))))) \\
\wedge \quad (((\mathit{r2} \in \mathbf{N}) \wedge ((\mathit{i0} * \mathit{i1}) = (\mathit{r2} + (\mathit{r0} * \mathit{r1}))) \wedge (\mathit{r0} \simeq 0)) \\
\qquad \rightarrow \quad (\mathit{r2} = (\mathit{i0} * \mathit{i1})))
\end{array}
$$

```
; Now we develop the analogue of the inductive assertion
; method formally.

; Introduce p as an arbitrary invariant under stepping.  The
; everywhere true predicate witnesses this constraint.
```

CONSERVATIVE AXIOM: p-step

$$\text{p}(s) \rightarrow \text{p}(\text{step}(s))$$

Simultaneously, we introduce the new function symbol p.

```
; Observe that such a p is invariant under arbitrary length runs of the
; machine.
```

THEOREM: p-invariant

$$\text{p}(\mathit{s0}) \rightarrow \text{p}(\text{sm}(\mathit{s0},\ n))$$

```
; That's it.  It is really deep isn't it?

; Now we'll define a p that suits our specification for TIMES.  We call
; it timesp.
```

DEFINITION: $\mathrm{r0}(s) = \mathrm{get}(\mathtt{0}, \mathrm{mem}(s))$

DEFINITION: $\mathrm{r1}(s) = \mathrm{get}(\mathtt{1}, \mathrm{mem}(s))$

DEFINITION: $\mathrm{r2}(s) = \mathrm{get}(\mathtt{2}, \mathrm{mem}(s))$

DEFINITION:
timesp (*i0*, *i1*, *s*)

$$
\begin{array}{l}
= \quad ((i0 \in \mathbf{N}) \\
\qquad \land \quad (i1 \in \mathbf{N}) \\
\qquad \land \quad \mathrm{stp}(s) \\
\qquad \land \quad (\mathrm{stk}(s) \simeq \mathbf{nil}) \\
\qquad \land \quad (\mathrm{assoc}(\texttt{'times}, \mathrm{defs}(s)) = \textsc{times-program}) \\
\qquad \land \quad (i1 = \mathrm{r1}(s)) \\
\qquad \land \quad \mathbf{if}\ \mathrm{pc}(s) = \texttt{'(times . 0)}\ \mathbf{then}\ i0 = \mathrm{r0}(s) \\
\qquad\qquad \mathbf{elseif}\ \mathrm{pc}(s) = \texttt{'(times . 1)} \\
\qquad\qquad \mathbf{then}\ (\mathrm{r2}(s) \in \mathbf{N}) \\
\qquad\qquad\qquad \land \quad ((i0 * i1) = (\mathrm{r2}(s) + (\mathrm{r0}(s) * \mathrm{r1}(s)))) \\
\qquad\qquad \mathbf{elseif}\ \mathrm{pc}(s) = \texttt{'(times . 2)} \\
\qquad\qquad \mathbf{then}\ (\mathrm{r0}(s) \not\simeq \mathtt{0}) \\
\qquad\qquad\qquad \land \quad (\mathrm{r2}(s) \in \mathbf{N}) \\
\qquad\qquad\qquad \land \quad ((i0 * i1) = (\mathrm{r2}(s) + (\mathrm{r0}(s) * \mathrm{r1}(s)))) \\
\qquad\qquad \mathbf{elseif}\ \mathrm{pc}(s) = \texttt{'(times . 3)} \\
\qquad\qquad \mathbf{then}\ (\mathrm{r0}(s) \not\simeq \mathtt{0}) \\
\qquad\qquad\qquad \land \quad (\mathrm{r2}(s) \in \mathbf{N}) \\
\qquad\qquad\qquad \land \quad ((i1 + (i0 * i1)) \\
\qquad\qquad\qquad\qquad = \quad (\mathrm{r2}(s) + (\mathrm{r0}(s) * \mathrm{r1}(s)))) \\
\qquad\qquad \mathbf{elseif}\ \mathrm{pc}(s) = \texttt{'(times . 4)} \\
\qquad\qquad \mathbf{then}\ (\mathrm{r2}(s) \in \mathbf{N}) \\
\qquad\qquad\qquad \land \quad ((i0 * i1) = (\mathrm{r2}(s) + (\mathrm{r0}(s) * \mathrm{r1}(s)))) \\
\qquad\qquad \mathbf{elseif}\ \mathrm{pc}(s) = \texttt{'(times . 5)}\ \mathbf{then}\ \mathrm{r2}(s) \\
\qquad\qquad\qquad\qquad\qquad\qquad = \quad (i0 * i1) \\
\qquad\qquad \mathbf{else}\ \mathbf{f}\ \mathbf{endif})
\end{array}
$$

```
; Since timesp is preserved by step:
```

THEOREM: timesp-step
$\mathrm{timesp}(i0, i1, s) \rightarrow \mathrm{timesp}(i0, i1, \mathrm{step}(s))$

```
; we can immediately conclude by functional instantiation that
; it is preserved under arbitrary runs of the machine:
```

THEOREM: timesp-invariant
$\text{timesp}(i0, i1, s0) \rightarrow \text{timesp}(i0, i1, \text{sm}(s0, n))$

```
; By additionally assuming that the initial and final pcs
; are at 0 and 5 respectively in TIMES, we derive the
; desired theorem.
```

THEOREM: times-correct-revisited-again

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{stp}(s0)} \\
\wedge & (\text{stk}(s0) \simeq \mathbf{nil}) \\
\wedge & (\text{assoc}(\texttt{'times}, \text{defs}(s0)) = \text{TIMES-PROGRAM}) \\
\wedge & (i0 = \text{get}(\texttt{0}, \text{mem}(s0))) \\
\wedge & (i1 = \text{get}(\texttt{1}, \text{mem}(s0))) \\
\wedge & (i0 \in \mathbf{N}) \\
\wedge & (i1 \in \mathbf{N}) \\
\wedge & (\text{pc}(s0) = \texttt{'(times . 0)}) \\
\wedge & (\text{pc}(\text{sm}(s0, n)) = \texttt{'(times . 5)})) \\
\rightarrow & (\text{get}(\texttt{2}, \text{mem}(\text{sm}(s0, n))) = (i0 * i1))
\end{array}
$$

```
; The following events are not at all easy to follow and should not be
; considered part of the tutorial.  They are included in this file to
; justify the sentence, in the second edition of the Handbook, that
; our standard form of correctness theorem for a subroutine implies
; the standard form of the termination theorem for that subroutine.
; In particular, we lead the system the proof of the following
; theorem.  Suppose s is a state poised to execute a CALL of some
; subroutine fn (and the halt flag of s is F).  Suppose that some
; non-zero number of steps, n, later the stack is the same as it is in
; s.  Intuitively, this means that the subroutine was called and
; eventually returned.  Then if the subroutine is called as the
; top-level program the halt flag is eventually set.  That is to say,
; let s' be obtained from s by setting the pc to (fn . 0), the first
; instruction in fn, and let the stack be nil, i.e., this is the
; top-level, main program.  Then by running s' n steps we obtain a
; state with the halt flag set.  That is the theorem
; standard-correctness-implies-termination, below.

; It is a fairly difficult theorem for two reasons.  First, it
; considers running fn in two different states: as part of a
; continuing computation and as the top-level main program.  We
```

```
; therefore have to develop lemmas that let us modify the state, e.g.,
; change the stack, without damaging some aspects of the computation.
; Second, the hypothesis that the stack eventually (at tick n) is the
; same as before the CALL means that a balanced RET was executed.  But
; it does not mean the balancing RET was executed at tick n.  For all
; we know, the CALL returned immediately and during the remaining
; ticks we possibly called other routines or even returned from the
; caller and eventually re-entered!  But we can convert that
; hypothesis into one that says for some k<n the balancing RET was
; executed on the kth tick and if we considered the top-level
; computation at that tick, we'll see that it sets the halt flag.  The
; remaining ticks at the top-level computation just leave the halt
; flag on.

; This proof took several days to construct and I found it frustrating
; in its complexity.  Perhaps someone can simplify it.  That said,
; here are the events with which I proved it.

; Because the tutorial has left the data base in a state designed to
; prove things of individual programs, there is a fair amount of
; enabling and disabling to get access to the guts of the machine.

; First we prove that once the machine halts, it stays halted.
```

THEOREM: step-preserves-haltedp
$(\neg\ \mathrm{haltedp}\,(\mathrm{step}\,(s))) \rightarrow (\neg\ \mathrm{haltedp}\,(s))$

THEOREM: sm-preserves-haltedp
$(\neg\ \mathrm{haltedp}\,(\mathrm{sm}\,(s,\ n))) \rightarrow (\neg\ \mathrm{haltedp}\,(s))$

```
; And that only RET sets the halt flag, i.e., if it becomes halted,
; then the current pc points to a RET.
```

THEOREM: only-ret-sets-haltedp
$((\neg\ \mathrm{haltedp}\,(s)) \wedge (\mathrm{haltedp}\,(\mathrm{step}\,(s)) \wedge (\mathit{defs} = \mathrm{defs}\,(s))))$
$\rightarrow \quad (\mathrm{car}\,(\mathrm{get}\,(\mathrm{cdr}\,(\mathrm{pc}\,(s)),\ \mathrm{cdr}\,(\mathrm{assoc}\,(\mathrm{car}\,(\mathrm{pc}\,(s)),\ \mathit{defs})))) = \texttt{'ret})$

```
; This function finds the k<n at which the balancing RET is
; executed.  Imagine that s is the state immediately after the
; CALL and that d is the depth of the stack in that state.
; Then we count ticks until we are poised to execute a RET from
; a state with stack depth d.
```

Definition:
k $(s,\ d,\ n)$
$=$ **if** $n \simeq$ `0` **then** `0`
elseif $(\text{length}(\text{stk}(s)) = d)$
$\wedge$ $(\text{car}(\text{fetch}(\text{pc}(s),\ \text{defs}(s))) =$ `'ret`$)$ **then** `0`
else $1 + \text{k}(\text{step}(s),\ d,\ n - 1)$ **endif**

```
; Because we'll keep step disabled, we'll need the following to
; analyze what it does to the stack depth.
```

Theorem: length-stk-step
$\text{length}(\text{stk}(\text{step}(s)))$
$=$ **if** $\text{haltedp}(s)$ **then** $\text{length}(\text{stk}(s))$
elseif $\text{car}(\text{fetch}(\text{pc}(s),\ \text{defs}(s))) =$ `'ret` **then** $\text{length}(\text{stk}(s)) - 1$
elseif $\text{car}(\text{fetch}(\text{pc}(s),\ \text{defs}(s))) =$ `'call` **then** $1 + \text{length}(\text{stk}(s))$
else $\text{length}(\text{stk}(s))$ **endif**

```
; The following theorem establishes that if, within n, the
; stack depth falls below d then the computed k is less than n.
```

Theorem: exists-terminating-ret
$((d \in \mathbf{N}) \wedge (\text{length}(\text{stk}(s0)) \not< d) \wedge (\text{length}(\text{stk}(\text{sm}(s0,\ n))) < d))$
$\rightarrow$ $(\text{k}(s0,\ d,\ n) < n)$

```
; We now want to prove that if the computed k is less than n, then
; various things are true of the state at tick k.  We need the
; obvious fact that the defs field never changes.
```

Theorem: defs-step
$\text{defs}(\text{step}(s)) = \text{defs}(s)$

```
; So here are some important properties of our k (when it is less than
; n), namely, that the stack depth of the kth state is d and that it
; is poised to execute a RET.
```

Theorem: properties-of-k
$(\text{k}(s0,\ d,\ n) < n)$
$\rightarrow$ $((\text{length}(\text{stk}(\text{sm}(s0,\ \text{k}(s0,\ d,\ n)))) = d)$
$\wedge$ $(\text{car}(\text{fetch}(\text{pc}(\text{sm}(s0,\ \text{k}(s0,\ d,\ n))),\ \text{defs}(s0))) =$ `'ret`$))$

```
; We also need that the kth state is still running, i.e., not itself halted.
; This takes a bit of work.
```

Theorem: haltedp-persists
$\text{haltedp}(s) \rightarrow \text{haltedp}(\text{sm}(s,\ n))$

Theorem: haltedp-k
$$
\begin{array}{l}
\text{haltedp}(s) \\
\rightarrow \quad (\text{k}(s,\ d,\ n) \\
\qquad = \quad \textbf{if}\ (\text{length}(\text{stk}(s)) = d) \\
\qquad\qquad \wedge \quad (\text{car}(\text{fetch}(\text{pc}(s),\ \text{defs}(s))) = \texttt{'ret})\ \ \textbf{then}\ \texttt{0} \\
\qquad\quad\ \textbf{else}\ \text{fix}(n)\ \textbf{endif})
\end{array}
$$

Theorem: halting-preserves-stk
$\text{haltedp}(\text{step}(s0)) \rightarrow (\text{length}(\text{stk}(\text{step}(s0))) = \text{length}(\text{stk}(s0)))$

```
; With that preamble, we can get that the kth state is still running.
```

Theorem: another-property-of-k
$$
\begin{array}{l}
((\neg\ \text{haltedp}(s0)) \wedge (\text{k}(s0,\ d,\ n) < n)) \\
\rightarrow \quad (\neg\ \text{haltedp}(\text{sm}(s0,\ \text{k}(s0,\ d,\ n))))
\end{array}
$$

```
; We assemble the two lemmas establishing properties of k into one: if
; s0 is not halted and within n ticks the stack is less than its
; current size then (a) k exists, i.e., is less than n, (b) the kth
; state has the same stack size as s0, (c) the kth state is poised to
; execute a RET and (d) it is not halted.
```

Theorem: decreasing-stk-means-ret-exists
$$
\begin{array}{l}
((\neg\ \text{haltedp}(s0)) \wedge (\text{length}(\text{stk}(\text{sm}(s0,\ n))) < \text{length}(\text{stk}(s0)))) \\
\rightarrow \quad ((\text{k}(s0,\ \text{length}(\text{stk}(s0)),\ n) < n) \\
\qquad \wedge \quad (\text{length}(\text{stk}(\text{sm}(s0,\ \text{k}(s0,\ \text{length}(\text{stk}(s0)),\ n)))) \\
\qquad\qquad = \quad \text{length}(\text{stk}(s0))) \\
\qquad \wedge \quad (\text{car}(\text{fetch}(\text{pc}(\text{sm}(s0,\ \text{k}(s0,\ \text{length}(\text{stk}(s0)),\ n))),\ \text{defs}(s0))) \\
\qquad\qquad = \quad \texttt{'ret}) \\
\qquad \wedge \quad (\neg\ \text{haltedp}(\text{sm}(s0,\ \text{k}(s0,\ \text{length}(\text{stk}(s0)),\ n)))))
\end{array}
$$

```
; Now we'll disable the two independently proved lemmas about k.
```

Event: Disable properties-of-k.

Event: Disable another-property-of-k.

```
; Next, we develop the idea that under some conditions we can mess around with
```

```
; the stack of a computation without changing the outcome in some sense.  The only
; way we'll mess around is by growing the stack at the deep end by adding some
; arbitrary additional cells.
```

DEFINITION:
grow-stk$(s,\ stk)$ = st(pc(s), append(stk(s), stk), mem(s), haltedp(s), defs(s))

```
; The lemma sm-grow-stk, just below, is the key result.  The intervening
; lemmas are just helpers.
```

THEOREM: listp-append
listp(append$(a,\ b)$) = (listp(a) $\vee$ listp(b))

THEOREM: step-grow-stk
($\neg$ haltedp(step(s)))
$\rightarrow$ (step(grow-stk$(s,\ stk)$) = grow-stk(step(s), stk))

THEOREM: sm-grow-stk
($\neg$ haltedp(sm$(s,\ n)$))
$\rightarrow$ (sm(grow-stk$(s,\ stk)$, n) = grow-stk(sm$(s,\ n)$, stk))

```
; The above lemma is really nice.  It says that if a computation
; doesn't halt within n then growing the stack commutes with the
; computation, i.e., you can grow the stack before you start or after
; you finish.  This lets us consider a computation in either of two
; states, one with a shallow stack or one with a deep stack.  If s has
; a stack of nil then it is in top-level execution and thus (grow-stk
; s stk) is some continuing execution of the same program.

; A key fact we'll need is that if k is less than or equal to n and the
; computation halts in k then it halts in n.  This explains why the
; halt flag is set at the end of the long top-level computation, even if
; it became set fairly early.
```

THEOREM: lessp-haltedp
$((n \not< k) \wedge$ haltedp(sm$(s,\ k)$)) $\rightarrow$ haltedp(sm$(s,\ n)$)

THEOREM: equal-length-0
(length$(x) = 0$) = $(x \simeq$ **nil**$)$

```
; Again, because step will be disabled later, we need to expose the
; behavior of a halting RET.
```

THEOREM: step-is-ret

$$((\neg \text{haltedp}(s)) \land (\text{car}(\text{fetch}(\text{pc}(s), \text{defs}(s))) = \texttt{'ret}) \land (\text{stk}(s) \simeq \textbf{nil})) \rightarrow \text{haltedp}(\text{step}(s))$$

```
; Oddly enough, though we proved that defs is preserved by step, above,
; we only now need that it is preserved by sm.
```

THEOREM: defs-sm

$$\text{defs}(\text{sm}(s, n)) = \text{defs}(s)$$

```
; In a sense, the following theorem is the real key to our proof.  It
; gives us a way to show that the halted flag is on in the nth step of
; s, namely find some k less than n-1 such that the kth state is not
; yet halted but has a stack of length 0 and is poised to execute a
; RET.  If you imagine that s is the top-level run of our subroutine,
; then this focusses our attention on the k at which the halt flag first
; becomes set.
```

THEOREM: expand-sm-n

$$((k < (n - 1)) \land (\neg \text{haltedp}(\text{sm}(s, k))) \land (\text{car}(\text{fetch}(\text{pc}(\text{sm}(s, k)), \text{defs}(s))) = \texttt{'ret}) \land (\text{length}(\text{stk}(\text{sm}(s, k))) = 0)) \rightarrow \text{haltedp}(\text{sm}(s, n))$$

```
; Now there are various details to be worked out, and I never found a
; really nice way to handle them except by brute force.  The basic
; theme of these details is that from the hypothesis that the
; ``continuing computation'' eventually returns to the same stack
; depth we can get some information about the pc and stack depth in
; the continuing computation.  But we have to convert that to
; information about the pc and stack depth in the top-level
; computation.  We can get these results from our sm-grow-stk lemma,
; namely, we know that if a short stacked computation doesn't halt we
; can grow its stack either before or after.  If the short stacked
; computation is the top-level one, where the stack is nil, then we
; can grow the stack to whatever stack we have in the continuing
; computation.  From the equality of the two final states we can learn
; that the pc of the top-level computation is the same as that of the
; continuing one.  While I find this proof very neat, what with its
; use of sm-grow-stk, I find the event below ugly because of the
```

```
; explict hint and the explicit states involved.  But it just wasn't
; worth my time to figure out an elegant rewrite rule that would
; normalize the pc.
```

THEOREM: pc-equiv

$(\neg\ \text{haltedp}\,(\text{sm}\,(\text{st}\,(\text{cons}\,(\textit{prog},\ 0),\ \textbf{nil},\ \text{mem}\,(s),\ \textbf{f},\ \text{defs}\,(s)),\ k)))$
$\rightarrow\quad (\text{pc}\,(\text{sm}\,(\text{st}\,(\text{cons}\,(\textit{prog},\ 0),\ \textbf{nil},\ \text{mem}\,(s),\ \textbf{f},\ \text{defs}\,(s)),\ k))$
$\quad = \quad \text{pc}\,(\text{sm}\,(\text{st}\,(\text{cons}\,(\textit{prog},\ 0),$
$\qquad\qquad \text{cons}\,(\text{cons}\,(\text{car}\,(\text{pc}\,(s)),\ 1 + \text{cdr}\,(\text{pc}\,(s))),\ \text{stk}\,(s)),$
$\qquad\qquad \text{mem}\,(s),$
$\qquad\qquad \textbf{f},$
$\qquad\qquad \text{defs}\,(s)),$
$\qquad k)))$

```
; We need to know a similar fact about the stacks after k steps.  In particular,
; we know from the continuing computation that at step k it is poised to RET on
; a stack of a certain depth.  We need to convert that to a fact about the top-level
; state at step k, namely that the stack there is nil -- so the RET will set the halt
; flag.  At first sight, this is a problem very similar to that above and one is
; tempted to try to solve it the same way.  But the problem above is insensitive to
; the value of k, as long as the computation is still running, while the one we
; are talking about now is our special k, the tick at which we execute the RET that
; balances the initial CALL.  But that raises a problem.  That existential k
; is computed with a given state.  Is that state from the continuing computation
; or from the top-level one?  What we prove below is that it doesn't matter, they
; are the same!  This is pretty subtle.  We need a few lemmas...
```

THEOREM: length-append

$\text{length}\,(\text{append}\,(a,\ b)) = (\text{length}\,(a) + \text{length}\,(b))$

THEOREM: grow-stk-props

$(\text{pc}\,(\text{grow-stk}\,(s,\ \textit{stk})) = \text{pc}\,(s))$
$\wedge\quad (\text{stk}\,(\text{grow-stk}\,(s,\ \textit{stk})) = \text{append}\,(\text{stk}\,(s),\ \textit{stk}))$
$\wedge\quad (\text{mem}\,(\text{grow-stk}\,(s,\ \textit{stk})) = \text{mem}\,(s))$
$\wedge\quad (\text{haltedp}\,(\text{grow-stk}\,(s,\ \textit{stk})) = \text{haltedp}\,(s))$
$\wedge\quad (\text{defs}\,(\text{grow-stk}\,(s,\ \textit{stk})) = \text{defs}\,(s))$

THEOREM: step-grow-stk-revisited-1

$(0 < \text{length}\,(\text{stk}\,(s)))$
$\rightarrow\quad (\text{step}\,(\text{grow-stk}\,(s,\ \textit{stk})) = \text{grow-stk}\,(\text{step}\,(s),\ \textit{stk}))$

THEOREM: step-grow-stk-revisited-2

$(\text{car}\,(\text{fetch}\,(\text{pc}\,(s),\ \text{defs}\,(s))) \neq \texttt{'ret})$
$\rightarrow\quad (\text{step}\,(\text{grow-stk}\,(s,\ \textit{stk})) = \text{grow-stk}\,(\text{step}\,(s),\ \textit{stk}))$

```
; So here is the key fact: k produces the same answer on the top-level
; state (here, s) and the continuing state, provided you bump the d
; appropriately.
```

THEOREM: k-grow-stk

$$((d \in \mathbf{N}) \land (\text{length}(\text{stk}(s)) \not< d)) \rightarrow (\text{k}(\text{grow-stk}(s,\ stk),\ d + \text{length}(stk),\ n) = \text{k}(s,\ d,\ n))$$

```
; Once again, I couldn't find a useful rewrite rule, since grow-stk isn't
; really in our problem, and so I make this lemma of class nil and instantiate
; it when I need to show that the two states produce the same k.  Given that,
; we can now infer that the final, top-level stack is nil at step k,
; just by using properties of k on the top-level state, but appealing to
; the existence of k from the continuing state.  We then repeat the exercise
; to extract the information that the halt flag is still off at step k in
; the top-level state.
```

THEOREM: stk-is-nil

$$\begin{aligned}
&(\text{k}(\text{st}(\text{cons}(prog,\ \mathtt{0}),\\
&\qquad \text{cons}(\text{cons}(\text{car}(\text{pc}(s)),\ 1 + \text{cdr}(\text{pc}(s))),\ \text{stk}(s)),\\
&\qquad \text{mem}(s),\\
&\qquad \mathbf{f},\\
&\qquad \text{defs}(s)),\\
&\quad 1 + \text{length}(\text{stk}(s)),\\
&\quad n - 1)\\
&\ < \ (n - 1))\\
&\rightarrow \ (\text{listp}(\text{stk}(\text{sm}(\text{st}(\text{cons}(prog,\ \mathtt{0}),\ \mathbf{nil},\ \text{mem}(s),\ \mathbf{f},\ \text{defs}(s)),\\
&\qquad\qquad \text{k}(\text{st}(\text{cons}(prog,\ \mathtt{0}),\\
&\qquad\qquad\qquad \text{cons}(\text{cons}(\text{car}(\text{pc}(s)),\ 1 + \text{cdr}(\text{pc}(s))),\ \text{stk}(s)),\\
&\qquad\qquad\qquad \text{mem}(s),\\
&\qquad\qquad\qquad \mathbf{f},\\
&\qquad\qquad\qquad \text{defs}(s)),\\
&\qquad\qquad\quad 1 + \text{length}(\text{stk}(s)),\\
&\qquad\qquad\quad n - 1))))\\
&\qquad = \ \mathbf{f})
\end{aligned}$$

THEOREM: haltedp-is-off

$$\begin{aligned}
&(\text{k}(\text{st}(\text{cons}(prog,\ \mathtt{0}),\\
&\qquad \text{cons}(\text{cons}(\text{car}(\text{pc}(s)),\ 1 + \text{cdr}(\text{pc}(s))),\ \text{stk}(s)),\\
&\qquad \text{mem}(s),\\
&\qquad \mathbf{f},\\
&\qquad \text{defs}(s)),\\
&\quad 1 + \text{length}(\text{stk}(s)),
\end{aligned}$$

$n - 1)$
$< \quad (n - 1))$
$\rightarrow \quad (\text{haltedp}\,(\text{sm}\,(\text{st}\,(\text{cons}\,(prog,\ \texttt{0}),\ \textbf{nil},\ \text{mem}\,(s),\ \textbf{f},\ \text{defs}\,(s)),$
$\quad \text{k}\,(\text{st}\,(\text{cons}\,(prog,\ \texttt{0}),$
$\quad\quad \text{cons}\,(\text{cons}\,(\text{car}\,(\text{pc}\,(s)),\ 1 + \text{cdr}\,(\text{pc}\,(s))),\ \text{stk}\,(s)),$
$\quad\quad \text{mem}\,(s),$
$\quad\quad \textbf{f},$
$\quad\quad \text{defs}\,(s)),$
$\quad 1 + \text{length}\,(\text{stk}\,(s)),$
$\quad n - 1)))$
$= \quad \textbf{f})$

```
; So, if you've followed all that, you are ready to get the main theorem:
```

THEOREM: standard-correctness-implies-termination
$((\neg\ \text{haltedp}\,(s))$
$\wedge \quad (\text{fetch}\,(\text{pc}\,(s),\ \text{defs}\,(s)) = \text{list}\,(\texttt{'call},\ prog))$
$\wedge \quad (n \not\simeq \texttt{0})$
$\wedge \quad (\text{stk}\,(\text{sm}\,(s,\ n)) = \text{stk}\,(s)))$
$\rightarrow \quad \text{haltedp}\,(\text{sm}\,(\text{st}\,(\text{cons}\,(prog,\ \texttt{0}),\ \textbf{nil},\ \text{mem}\,(s),\ \textbf{f},\ \text{defs}\,(s)),\ n))$

```
; As I said, the proof is not at all easy to follow.  I invite
; readers to find a better one!


; The next theorem establishes the effect of a one-instruction infinite loop.
; It says that if you have a running state and when you fetch the current
; instruction you get (JUMP i) where i is the location of the current program
; counter, then the halt flag is never set.
```

THEOREM: infinite-loop
$((\neg\ \text{haltedp}\,(s))$
$\wedge \quad (\text{fetch}\,(\text{pc}\,(s),\ \text{defs}\,(s)) = \text{list}\,(\texttt{'jump},\ i))$
$\wedge \quad (i \in \mathbf{N})$
$\wedge \quad (\text{cdr}\,(\text{pc}\,(s)) = i))$
$\rightarrow \quad (\neg\ \text{haltedp}\,(\text{sm}\,(s,\ n)))$

Index

add, 3, 4
add1-pc, 2–4, 7, 10, 13
another-property-of-k, 19
associativity-of-plus, 12
associativity-of-times, 9

call, 4
commutativity-of-plus, 12
commutativity2-of-plus, 12

decreasing-stk-means-ret-exists, 19
defs, 3–5, 13–24
defs-sm, 21
defs-step, 18
difference-1, 11
difference-elim, 12

equal-length-0, 20
execute, 4, 5
exists-terminating-ret, 18
exp, 8–10
exp-clock, 9, 10
exp-correct, 10
exp-correct-lemma, 10
exp-fn, 9
exp-fn-is-exp, 9
exp-program, 8, 10
expand-sm-n, 21

fetch, 3–5, 7, 10, 14, 18, 19, 21, 22, 24

get, 2–4, 11–17
get-put, 11
grow-stk, 20, 22, 23
grow-stk-props, 22

haltedp, 4, 5, 14, 17–22, 24
haltedp-is-off, 23
haltedp-k, 19
haltedp-persists, 19
halting-preserves-stk, 19

infinite-loop, 24

jump, 4
jumpz, 4

k, 18, 19, 23, 24
k-grow-stk, 23

length, 11, 12, 14, 18–24
length-append, 22
length-put, 11
length-stk-step, 18
lessp-haltedp, 20
listp-append, 20

mem, 3, 4, 13–16, 20, 22–24
move, 3, 4
movi, 3, 4

only-ret-sets-haltedp, 17

p, 14
p-invariant, 14
p-step, 14
pc, 3–5, 13–24
pc-equiv, 22
plus-right-id, 7
properties-of-k, 18
put, 3, 11–13
put-put-0, 11
put-put-1, 11
put-put-2, 11

r0, 15
r1, 15
r2, 15
ret, 4

sm, 5, 7, 10, 13, 14, 16–24
sm-0, 5
sm-add1, 5
sm-grow-stk, 20

sm-plus, 5
sm-preserves-haltedp, 17
st, 2–4, 7, 10, 13, 20, 22–24
standard-correctness-implies-te
rmination, 24
step, 4, 5, 14, 15, 17–22
step-grow-stk, 20
step-grow-stk-revisited-1, 22
step-grow-stk-revisited-2, 22
step-is-ret, 21
step-opener, 5
step-preserves-haltedp, 17
stk, 3, 4, 13, 15, 16, 18–24
stk-is-nil, 23
stp, 15, 16
subi, 3, 4

times-clock, 7, 9, 10, 14
times-correct, 7
times-correct-lemma, 7
times-correct-lemma-revisited, 13
times-correct-lemma-revisited-a
nd-generalized, 13
times-correct-revisited, 14
times-correct-revisited-again, 16
times-fn, 6, 7
times-fn-is-times, 6
times-mem-fn, 12
times-mem-fn-is-correct, 12
times-mem-fn-loop, 12, 13
times-mem-fn-loop-is-times, 12
times-program, 6, 7, 10, 13–16
times-right-id, 9
times-step, 13, 14
timesp, 15, 16
timesp-invariant, 16
timesp-step, 15

verification-conditions-for-time
s, 14