Combining an Interpeter-Based Approach to Software Verification with Verification Condition Generation

Matt Kaufmann

Technical Report 97 April, 1994

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

ABSTRACT: *Two traditional approaches to software verification are an* interpreter-based *approach and an approach using a* verification conditional generator *(*VCG*). The former approach involves proving the program correct directly from the language definition, while the latter uses an auxiliary tool (a verification conditional generator) to generate proof obligations. We report here on some preliminary work that ties these two approaches together, thus eliminating the need to trust a VCG while still enjoying the kind of assistance that a VCG supplies.*

1. INTRODUCTION

Two traditional approaches to software verification are an interpreter-based approach and an approach using a verification conditional generator (VCG). The interpreter-based approach is to define an operational-style semantics for the language, and prove that the interepreter behaves properly on a given program with given inputs, under suitable hypotheses. The latter approach is based on the Floyd-Hoare approach to program verification, using a so-called VCG, to generate proof obligations from the given program, which has been annotated with input and output assertions together with possible additional annotations, such as loop invariants. The former approach enjoys the property that since the program has been verified directly against the language definition, one need only trust that definition (together, of course, with the program's specification and the theorem prover used to carry out the reasoning if such an automated tool is employed) in order to trust the result. The latter approach traditionally relies on an untrusted (unverified) VCG, but seems to be much easier in general to carry out for particular programs; this point was made for example in [1].

The present work reports some early successes in tying together these two approaches. Ultimately, the verification of a program with the approach in this report is one that fits into the first paradigm above, and thus enjoys the assurance and flexibility that one obtains by reasoning directly from the semantics. However, we develop a tool much like a VCG and employ it to ease the burden of such a proof. The technique of functional instantiation (see [2]) is used to tie together the VCG approach with the operational semantics. As a result, we do not actually rely on the correctness of our VCG-like tool; rather, it is simply a heuristic device that helps us with our interpreter-based proofs.

The first section below introduces a toy programming language interpreter that we use throughout our development. The second section reviews the notion of proving loops directly with an interpreter by using a particular very simple example. The third section shows how we introduce a very abstract loop, prove it correct, and functionally instantiate the resulting theorems to provide an alternate proof of the theorem proved in the preceding section. Next, we discuss the tools that we have developed for automating the sort of proof described in the preceding section. Finally, we discuss some early work in extending our approach from loops to procedure calls.

All of our definitions are written in Acl2 [3]. In this paper we use an infix-style syntax when it seems appropriate. The Acl2 sources may be found in appendices, as follows.

- Appendix A *(basic.lisp)* The interpreter for our toy language, together with a small but essential lemma base;
- Appendix B *(generic.lisp)* the ‘‘generic’’ loop and procedure code and lemmas about them, for functional instantiation.
- Appendix C *(force.lisp)* A theory containing lemmas with **`force`**d hypotheses, particularly for use in a symbolic evaluator defined in *macros.lisp*.
- Appendix D *(macros.lisp)* Definition of our analogue of a VCG, **`defprogram`**.

2. A TOY PROGRAMMING LANGUAGE INTERPRETER

The usual interpreter-based approach is to define a function **`interpret(program, state)`** that takes a program in the language and a program state and returns a new program state. For our toy language, we will use as the state a *variable alist*, which is a list that associates values with variables. That is, a variable alist is a list of pairs **`(variable . value)`**; the meaning of such a pair is that the value of **`variable`** is **`value`**. In our toy language, all values are integers.

The notion of looking up the value of a variable is defined in the usual way. The function **`variable-value`** is similar to the Lisp function **`assoc`**, except that it's defined so that no guard needs to be supplied.

DEFINITION:
variable-value (*x*, *va*)
=
if atom (*va*) **then nil**
elseif consp (car (*va*)) ∧ *x* = caar (*va*) **then** cdar (*va*)
else variable-value (*x*, cdr (*va*))
endif

Accompanying the notion of the value of a variable is the notion of updating the state. We'll need this notion when we define the semantics of an assignment statement.

DEFINITION:
variable-update (*var*, *val*, *va*)
=
if consp (*va*)
then if consp (car (*va*)) ∧ caar (*va*) = *var*
    then cons (cons (*var*, *val*), cdr (*va*))
    else cons (car (*va*), variable-update (*var*, *val*, cdr (*va*)))
    endif
else *va*
endif

Before proceeding further, we need the notion of a variable being *bound* in a given variable alist, i.e., the notion of a variable being associated with some value in that variable alist.

DEFINITION:
variable-boundp (*var*, *va*)
=
if consp (*va*)
then if consp (car (*va*)) ∧ caar (*va*) = *var* **then t**
    else variable-boundp (*var*, cdr (*va*))
    endif
else nil
endif

The next major step is to define an expression evaluator. There are various auxiliary definitions, including the following. <* is like ordinary < except that it returns 1 for *true* and 0 for *false*:

DEFINITION:
<* (*x*, *y*)
=
if fix-int (*x*) < fix-int (*y*) **then `1`**
else `0`
endif

The expression evaluator is then defined as follows.

DEFINITION:
eval-exp (*list-flag*, *x*, *va*)
=
if *list-flag*
then if consp (*x*)
    then cons (eval-exp (**nil**, car (*x*), *va*), eval-exp (**t**, cdr (*x*), *va*))
    else nil

```
        endif
else if integerp (x)  then x
       elseif atom (x)  then variable-value (x, va)
       elseif car (x) = 'plus
       then     fix-int (eval-exp (nil, safecar (cdr (x)), va))
              +  fix-int (eval-exp (nil, safecar (safecdr (cdr (x))), va))

       elseif car (x) = 'negative
       then fix-int (eval-exp (nil, safecar (cdr (x)), va))
       elseif car (x) = 'times
       then     fix-int (eval-exp (nil, safecar (cdr (x)), va))
              ×  fix-int (eval-exp (nil, safecar (safecdr (cdr (x))), va))

       elseif car (x) = 'lessp
       then <* (fix-int (eval-exp (nil, safecar (cdr (x)), va)),
                fix-int (eval-exp (nil, safecar (safecdr (cdr (x))), va)))
       else 0
       endif
endif
```

There are a number of useful lemmas to prove in order to use our expression evaluator and statement interpreter. Here is an example, which characterizes the value of a variable in the result of updating a variable alist.. Again, a complete input file for Acl2 may be found in Appendix A.

THEOREM: variable-value-variable-update

```
     variable-value (x, variable-update (y, val, va))
=    if x = y
      then if variable-boundp (x, va)  then val
               else nil
               endif
      else variable-value (x, va)
      endif
```

We are now ready to define the interpreter. The definition has the following form. Note that **`interpret`** returns a list of two values, **`(mv exc val)`**, where **`exc`** is **`t`** if the input clock was not sufficient to allow deeply nested calls of the interpreter and **`nil`** otherwise, and **`val`** is the new variable alist (in the case that **`exc`** is **`nil`**.

DEFINITION:

```
interpret (flg, stmt, va, pa, clock)
 =
if flg  then
  < This is the case where we have a list of statements. >
 elseif <assignment_statement>  then
  (mv nil <update the variable alist>)
 elseif <whileloop_statement>  then
  < (while test . statements):  evaluate the test, and if non-zero, then
     evaluate the statements and recur >
 elseif <if_statement>  then
  < Run interpreter recursively on true or false branch >
 elseif <let*_statement>  then
  < Bind locals to their values, and continue >
 elseif <procedure_call>  then
  < Bind the formals to the actuals, interpret the body, and then copy out
     the appropriate new values if there is no exception. >
 else { nil, va }
 endif
```

Here is the entire definition, which fleshes out the outline above.

```
DEFINITION:
interpret (flg, stmt, va, pa, clock)
 =
if flg
 then if consp (stmt)
         then { exc, va } := interpret (nil, car (stmt), va, pa, clock);

                 if exc  then { exc, va }
                  else interpret (t, cdr (stmt), va, pa, clock)
                  endif

         else { nil, va }
         endif
 elseif consp (stmt) ∧ car (stmt) = 'assign
 then { nil, variable-update (safecar (cdr (stmt)),
                                          eval-exp (nil, safecar (safecdr (cdr (stmt))), va),
                                          va) }
 elseif consp (stmt) ∧ car (stmt) = 'while
 then if eval-exp (nil, safecar (cdr (stmt)), va) = 0  then { nil, va }
         elseif fix-int (clock) ≤ 0  then { t, va }
         else { exc, new-va } := interpret (t, safecdr (cdr (stmt)), va, pa, clock);

                if exc  then { exc, va }
                 else interpret (nil, stmt, new-va, pa, clock − 1)
                 endif

         endif
 elseif consp (stmt) ∧ car (stmt) = 'if
 then if eval-exp (nil, safecar (cdr (stmt)), va) = 0
         then interpret (nil,
                              safecar (safecdr (safecdr (cdr (stmt)))),
                              va,
                              pa,
                              clock)
         else interpret (nil, safecar (safecdr (cdr (stmt))), va, pa, clock)
         endif
 elseif consp (stmt) ∧ car (stmt) = 'let*
 then { exc, new-va } := interpret (t,
                                              safecdr (cdr (stmt)),
                                              declare-locals (safecar (cdr (stmt)), va),
                                              pa,
                                              clock);

        { exc, popn (len (safecar (cdr (stmt))), new-va) }

 elseif consp (stmt) ∧ car (stmt) = 'proc-call
 then if fix-int (clock) ≤ 0  then { t, va }
         else let proc be proc-lookup (safecar (cdr (stmt)), pa),
                    actuals be safecdr (cdr (stmt)) in
               { exc, post-va } := interpret (t,
                                                      proc-body (proc),
                                                      bind-formals (proc-formals (proc),
                                                                          eval-exp (t,
                                                                                        actuals,
                                                                                        va)),
                                                      pa,
                                                      clock − 1);
```

```
        if exc  then { exc, va }
         else { nil, copy-out (proc-formals (proc),
                               proc-outlist (proc),
                               actuals,
                               post-va,
                               va) }
         endif

    endif
 else { nil, va }
 endif
Measure:cons (1 +if 0 ≤ fix-int (clock)  then fix-int (clock)
                else 0
                endif,
            acl2-count (stmt))
```

3. PROVING A LOOP CORRECT USING A Toy Programming Language Interpreter

Consider the following very simple loop written in our toy language. Note that a WHILE loop terminates when its test is 0.

```
WHILE X
  X := X - 1
  Y := Y + 1
```

Actually, for completeness we should note that the official syntax for such a loop looks as follows. However, we will not consider that syntax further below.

```
(while x
  (assign x (plus x -1))
  (assign y (plus y 1)))
```

A reasonable invariant (needed to guarantee termination) is that **X** is not negative. Our specification of this loop is that if the invariant holds on entry to the loop, then the final values **X’** and **Y’** of **X** and **Y** have the following values:

```
  X’ := 0
  Y’ := X + Y
```

One can take the following standard approach to proving correctness of such a loop. Let us call our specification function **add$spec-explicit**. Thus, **add$spec-explicit** maps a variable alist to the effect of assigning **X** to 0 and **Y** to the sum of the preexisting values of **X** and **Y**.

Step 1. Specify the effect of passing one time through the loop. (This can generally be done automatically.)

```
  X := X - 1
  Y := Y + 1
```

Let us denote the variable-alist resulting from stepping *va* once through the loop as **add$body-spec**(*va*).

Step 2. Prove that the invariant holds each time we pass through the loop.

```
invariant(va) -> invariant(body-spec(va))
```

Step 3. Define a measure **`add$measure`** that guarantees termination of the loop, assuming that the invariant holds. We may define **`add$measure(`***va***`)`** to be essentially the value of the test of the loop, i.e., the value of the program variable **`X`** in the environment *va*.

Step 4. Define an ‘‘obvious’’ recursive function **`add$spec`** that maps a variable-alist *va* to the result of running the loop on it. Use the measure from Step 3 to prove termination.

DEFINITION:
add$spec (*va*)
=
if ¬ add$invariant (*va*) ∨ eval-exp (**nil**, **add$test**, *va*) = **`0`** **then** *va*
else add$spec (add$body-spec (*va*))
endif
Measure:add$measure (*va*)

Step 5. Prove that the ‘‘obvious’’ recursive function **`add$spec`** is a correct characterization of the effect of running the loop on a given environment.

THEOREM: add$correct
add$invariant (*va*) ∧ ¬ car (interpret (**nil**, **add**, *va*, *pa*, *clock*))
→ interpret (**nil**, **add**, *va*, *pa*, *clock*) = list (**nil**, add$spec (*va*))

Step 6. Prove a key lemma for use in the inductive step of the proof in Step 7.

THEOREM: key-step
eval-exp (**nil**, **add$test**, *va*) ≠ **`0`**
∧ add$invariant (*va*)
∧ *va2* = add$body-spec (*va*)
→ add$spec-explicit (*va2*) = add$spec-explicit (*va*)

Step 7. Prove the equivalence of the ‘‘obvious’’ recursive function with the specification function, by induction.

Step 8. Conclude that the specification function correctly characterizes the effect of the loop, as an immediate consequences of Steps 5 and 7.

4. Proving a Loop Correct Using a Toy Programming Language Interpreter BY FUNCTIONAL INSTANTIATION

Time does not permit a detailed explanation of this work; for that I refer you to CLI Technical Report 97. Roughly, our approach is to carry out the steps described above for the add loop, but for *constrained* functions in place of particular functions. The Acl2 functional instantiation mechanism then lets us derive correctness of specific programs from the general case, assuming that we can prove the appropriate functional instances of the defining axioms. This amounts to proving the main steps outlined for the addition loop above, *except* that the non-specific ‘‘glue’’ steps may be avoided, including the proof by induction in Step 7.

5. AUTOMATING Proving a Loop Correct Using a Toy Programming Language Interpreter by Functional Instantiation

I have written a fairly elaborate utility called **`defprogram`**, COMPLETELY in Acl2, that takes a program and its specification and writes out an event file. This file is suitable as input to Acl2, and is intended to be processed fully automatically. However, the key lemma (Step 6 above) may not be proved fully automatically, in which case the user can edit that file until the proofs all succeed.

Example invocation of **`defprogram`**:

```
(DEFPROGRAM
  ADD
  '(WHILE X
          (ASSIGN X (PLUS X -1))
          (ASSIGN Y (PLUS Y 1)))
  :INVARIANT (<= 0 (@PRE X))
  :EXIT
  (@PASSIGN (X 0)
            (Y (+ (@PRE X) (@PRE Y)))))
```

Other loops proved correct using this approach:

- Multiplication by successive addition (nested loops)
- Greatest common divisor
- Integer logarithm base 2

6. EXTENDING THIS WORK TO PROCEDURE CALLS

I have used functional instantiation to prove the correctness of a multiplication *procedure* whose body has the following abstract syntax.

```
(mult (a b ans)       ;parameter list
      (ans)           ;out (''var'', writable) parameter sublist
      (let*           ;''block'' statement
          ;locals
          ((x 0)
           (y 0)
           (counter a)
           (v b))
          ;statements
        <nested while loop>
        (assign ans y)))
```

This work is, however, very preliminary.

Appendix A

interpreter

The file *interpreter.lisp* contains the interpreter for our toy language, together with a small but essential lemma base.

```
;;; A very simple semantics with an out-of-time exception.

(in-package "ACL2")

;;; Include a book of facts about rational numbers.

(include-book "rationals-1/top")

;;; This work was originally done in version 1.4 of Acl2, where nth
;;; was enabled.

(in-theory (enable nth))

;;; 0 plays the role for IF* that NIL does for IF.  We define IF*,
;;; prove the two rules that characterize it, and then disable it.
;;; With this approach we are guaranteeing that IF* will not cause
;;; case splits.

(defun if* (x y z)
  (if (equal x 0) z y))

(defthm if*-non-zero
  (implies (not (equal x 0))
           (equal (if* x y z)
                  y)))

(defthm if*-zero
  (equal (if* 0 y z)
         z))

(in-theory (disable if*))

(defun fix-int (x)
  (if (integerp x)
      x
    0))

;;; <* is like <, but returns 1 for true and 0 for false.  We prove
;;; some important rules about <*, and then disable it so that we
;;; prevent it from causing case splits.

(defun <* (x y)
  (if (< (fix-int x) (fix-int y))
      1
    0))

(defthm equal-<*-0
  (equal (equal (<* x y) 0)
         (not (< (fix-int x) (fix-int y)))))

(defthm <*-zero
  (implies (< (fix-int x) (fix-int y))
           (equal (<* x y)
                  1)))

(defthm <*-non-zero
  (implies (not (< (fix-int x) (fix-int y)))
           (equal (<* x y)
                  0)))
```

```
(in-theory (disable <*))

;;; We would prefer, for this small experiment, to avoid dealing with
;;; guards (at least on this pass).  Therefore we introduce "safe"
;;; versions of car and cdr, prove some important properties of them,
;;; and disable them (thus avoiding case splits).

(defun safecar (x)
  (and (consp x) (car x)))

(defun safecdr (x)
  (and (consp x) (cdr x)))

(defthm safecar-consp
  (implies (consp x)
           (equal (safecar x)
                  (car x))))

(defthm safecdr-consp
  (implies (consp x)
           (equal (safecdr x)
                  (cdr x))))

(defthm safecar-count
  (<= (acl2-count (safecar x))
      (acl2-count x))
  :rule-classes :linear)

(defthm safecdr-count
  (<= (acl2-count (safecdr x))
      (acl2-count x))
  :rule-classes :linear)

(in-theory (disable safecar safecdr))

;;; In the next three definitions, the conjunct (consp (car va)) is
;;; there because we want to avoid dealing with guards on this pass.

(defun variable-value (x va)
  (cond ((atom va)
         ;; We could return 0 instead of nil here, since our toy
         ;; language involves only the integer data type.  However, we
         ;; return nil with an eye towards scaling up our approach.
         nil)
        ((and (consp (car va))
              (equal x (caar va)))
         (cdar va))
        (t (variable-value x (cdr va)))))

(defun variable-update (var val va)
  (if (consp va)
      (if (and (consp (car va))
               (equal (caar va) var))
          (cons (cons var val)
                (cdr va))
        (cons (car va)
              (variable-update var val (cdr va))))
    ;; need va below instead of nil, in order to prove
    ;; variable-update-no-op below
    va))

(defun variable-boundp (var va)
  (if (consp va)
      (if (and (consp (car va))
               (equal (caar va) var))
          t
        (variable-boundp var (cdr va)))
    nil))
```

```
;;; Next we prove crucial lemmas, both interesting ones and
;;; straigtforward ‘‘opener’’ lemmas, before disabling these
;;; definitions.

(defthm variable-update-no-op
  (implies (equal val (variable-value var va))
           (equal (variable-update var val va)
                  va)))

;;; It’s tempting to create a single rewrite rule in place of the
;;; following two, namely

#|
(defthm variable-update-variable-update
  (equal (variable-update x1 val1
                          (variable-update x2 val2
                                           va))
         (if (equal x1 x2)
             (variable-update x1 val1 va)
             (variable-update x2 val2
                              (variable-update x1 val1
                                               va))))
  :rule-classes
  ((:rewrite :loop-stopper ((x1 x2)))))
|#

;;; However, that won’t work, because the loop-stopper algorithm
;;; checks strict lexicographic term-order; thus, the case where x1
;;; and x2 are syntactically the same will not allow the rule to be
;;; applied.  Therefore, here instead are two separate rules.

(defthm variable-update-variable-update
  (implies (not (equal x1 x2))
           (equal (variable-update x1 val1
                                   (variable-update x2 val2
                                                    va))
                  (variable-update x2 val2
                                   (variable-update x1 val1
                                                    va))))
  :rule-classes
  ((:rewrite :loop-stopper ((x1 x2)))))

(defthm variable-update-variable-update-equal
  (equal (variable-update x1 val1
                          (variable-update x1 val2 va))
         (variable-update x1 val1 va)))

(defthm variable-value-variable-update
  (equal (variable-value x (variable-update y val va))
         (if (equal x y)
             (if (variable-boundp x va)
                 val
               nil)
           (variable-value x va))))

(defthm variable-boundp-variable-update
  (equal (variable-boundp u (variable-update v val va))
         (variable-boundp u va)))

(defthm variable-value-opener
  (and (equal (variable-value x nil)
              nil)
       (equal (variable-value x (cons (cons x val) va))
              val)
       (implies (not (equal x y))
                (equal (variable-value x (cons (cons y val) va))
                       (variable-value x va)))))

(defthm variable-update-opener
```

```
  (and (equal (variable-update var val nil)
              nil)
       (equal (variable-update var val
                               (cons (cons var old-val) va))
              (cons (cons var val) va))
       (implies (not (equal var1 var2))
                (equal (variable-update var1 val1
                                        (cons (cons var2 val2) va))
                       (cons (cons var2 val2)
                             (variable-update var1 val1 va))))))

(defthm variable-boundp-opener
  (and (equal (variable-boundp var nil)
              nil)
       (equal (variable-boundp var1 (cons (cons var2 val) va))
              (or (equal var1 var2)
                  (variable-boundp var1 va)))))

(in-theory (disable variable-update variable-boundp variable-value))

;;; It's not necessary to have a ''list-flag'' below, but we do so
;;; with an eye towards more complicated expression languages, in
;;; which operators can take arbitrary-length lists of arguments.
;;; After giving the obvious opener lemmas, we disable this function.

(defun eval-exp (list-flag x va)
  (if list-flag
      (if (consp x)
          (cons (eval-exp nil (car x) va)
                (eval-exp t (cdr x) va))
        nil)
    (cond
     ((integerp x) x)
     ((atom x)
      (variable-value x va))
     ((equal (car x) 'plus)
      (+ (fix-int (eval-exp nil (safecar (cdr x)) va))
         (fix-int (eval-exp nil (safecar (safecdr (cdr x))) va))))
     ((equal (car x) 'negative)
      (- (fix-int (eval-exp nil (safecar (cdr x)) va))))
     ((equal (car x) 'times)
      (* (fix-int (eval-exp nil (safecar (cdr x)) va))
         (fix-int (eval-exp nil (safecar (safecdr (cdr x))) va))))
     ((equal (car x) 'lessp)
      (<* (fix-int (eval-exp nil (safecar (cdr x)) va))
          (fix-int (eval-exp nil (safecar (safecdr (cdr x))) va))))
     (t 0))))

(defthm eval-exp-list
  (and (equal (eval-exp t nil va)
              nil)
       (equal (eval-exp t (cons a x) va)
              (cons (eval-exp nil a va)
                    (eval-exp t x va)))))

(defthm eval-exp-integer
  (implies (integerp x)
           (equal (eval-exp nil x va)
                  x)))

(defthm eval-exp-atom
  (implies (and (not (integerp x))
                (atom x))
           (equal (eval-exp nil x va)
                  (variable-value x va))))

(defthm eval-exp-plus
  (implies
   (and (consp x)
```

```
        (equal (car x) 'plus))
   (equal
    (eval-exp nil x va)
    (+ (fix-int (eval-exp nil (safecar (cdr x)) va))
       (fix-int (eval-exp nil (safecar (safecdr (cdr x))) va))))))

(defthm eval-exp-negative
  (implies (and (consp x)
                (equal (car x) 'negative))
           (equal (eval-exp nil x va)
                  (- (fix-int (eval-exp nil (safecar (cdr x)) va))))))

(defthm eval-exp-times
  (implies
   (and (consp x)
        (equal (car x) 'times))
   (equal
    (eval-exp nil x va)
    (* (fix-int (eval-exp nil (safecar (cdr x)) va))
       (fix-int (eval-exp nil (safecar (safecdr (cdr x))) va))))))

(defthm eval-exp-lessp
  (implies
   (and (consp x)
        (equal (car x) 'lessp))
   (equal
    (eval-exp nil x va)
    (<* (fix-int (eval-exp nil (safecar (cdr x)) va))
        (fix-int (eval-exp nil (safecar (safecdr (cdr x))) va))))))

(in-theory (disable eval-exp))

;;; There are a number of definitions to make for defining the
;;; semantics of procedure calls.  As usual, we disable these
;;; functions after proving appropriate opener lemmas.

(defun proc-lookup (proc-name pa)
  (if (consp pa)
      ;; (car pa) should be (name args outlist . statements)
      (if (and (true-listp (car pa))
               (equal proc-name (caar pa)))
          (car pa)
        (proc-lookup proc-name (cdr pa)))
    nil))

(defthm proc-lookup-type
  (true-listp (proc-lookup name alist))
  :rule-classes :type-prescription)

(defthm proc-lookup-opener
  (implies (true-listp x)
           (equal (proc-lookup proc-name
                               (cons (cons name x) pa))
                  (if (equal proc-name name)
                      (cons name x)
                    (proc-lookup proc-name pa)))))

(defmacro proc-name (proc)
  `(car ,proc))

(defmacro proc-formals (proc)
  `(cadr ,proc))

(defmacro proc-outlist (proc)
  `(caddr ,proc))

(defmacro proc-body (proc)
  `(cdddr ,proc))
```

```
(defun bind-formals (formals vals)
  (if (consp formals)
      (cons (cons (car formals)
                  (safecar vals))
            (bind-formals (cdr formals) (safecdr vals)))
    nil))

(defthm bind-formals-opener
  (and (equal (bind-formals nil vals)
              nil)
       (equal (bind-formals (cons formal formals)
                            (cons val vals))
              (cons (cons formal val)
                    (bind-formals formals vals)))))

;;; Here is a no-guard version of member-equal.

(defun member/ (a lst)
  (if (consp lst)
      (or (equal a (car lst))
          (member/ a (cdr lst)))
    nil))

(defun copy-out (formals outlist actuals post-va va)
  (if (consp formals)
      (if (member/ (car formals) outlist)
          (copy-out (cdr formals) outlist (safecdr actuals) post-va
                    (variable-update (safecar actuals)
                                     (variable-value (car formals) post-va)
                                     va))
        (copy-out (cdr formals) outlist (safecdr actuals) post-va va))
    va))

(defthm copy-out-opener
  (and (equal (copy-out nil outlist actuals post-va va)
              va)
       (implies (member/ formal outlist)
                (equal (copy-out (cons formal formals)
                                 outlist
                                 (cons actual actuals)
                                 post-va va)
                       (copy-out formals outlist actuals post-va
                                 (variable-update
                                  actual
                                  (variable-value formal post-va)
                                  va))))
       (implies
        (not (member/ formal outlist))
        (equal (copy-out (cons formal formals)
                         outlist
                         (cons actual actuals)
                         post-va va)
               (copy-out formals outlist actuals post-va va)))))

;;; Note that we'll only open member/ on explicit values, so it
;;; suffices to leave only the executable counterpart of member/
;;; enabled.

(in-theory (disable proc-lookup bind-formals member/ copy-out))

(defun declare-locals (doublets va)
  (if (consp doublets)
      (declare-locals
       (cdr doublets)
       (cons (cons (safecar (car doublets))
                   (eval-exp nil
                             (safecar (safecdr (car doublets)))
                             va))
             va))
```

```
    va))

(defthm declare-locals-opener
  (and (equal (declare-locals nil va)
              va)
       (equal (declare-locals (cons (list var x) locals) va)
              (declare-locals locals
                              (cons (cons var (eval-exp nil x va))
                                    va)))))

(defun popn (n lst)
  (if (< 0 (fix-int n))
      (popn (+ -1 n) (safecdr lst))
    lst))

(defthm popn-opener
  (and (implies (< 0 (fix-int n))
                (equal (popn n lst)
                       (popn (+ -1 n) (safecdr lst))))
       (implies (not (< 0 (fix-int n)))
                (equal (popn n lst)
                       lst))))

(in-theory (disable declare-locals popn))

;;; Finally, here is our statement interpreter.

(defun interpret (flg stmt va pa clock)
  (declare (xargs
            :measure
            (cons (+ 1 (if (<= 0 (fix-int clock)) (fix-int clock) 0))
                  (acl2-count stmt))))
  (cond
   (flg
    (if (consp stmt)
        (mv-let (exc va)
            (interpret nil (car stmt) va pa clock)
          (if exc
              (mv exc va)
            (interpret t (cdr stmt) va pa clock)))
      (mv nil va)))
   ((and (consp stmt)
         (equal (car stmt) 'assign))
    (mv nil (variable-update
             (safecar (cdr stmt))
             (eval-exp nil (safecar (safecdr (cdr stmt))) va)
             va)))
   ((and (consp stmt)
         (equal (car stmt) 'while))
    (if (equal (eval-exp nil (safecar (cdr stmt)) va) 0)
        (mv nil va)
      (if (<= (fix-int clock) 0)
          (mv t va)
        (mv-let (exc new-va)
            (interpret t (safecdr (cdr stmt)) va pa clock)
          (if exc
              (mv exc va)
            (interpret nil stmt new-va pa (- clock 1)))))))
   ((and (consp stmt)
         (equal (car stmt) 'if))
    (if (equal (eval-exp nil (safecar (cdr stmt)) va) 0)
        (interpret nil (safecar (safecdr (safecdr (cdr stmt))))
                   va pa clock)
      (interpret nil (safecar (safecdr (cdr stmt))) va pa clock)))
   ((and (consp stmt)
         (equal (car stmt) 'let*))
    ;; same syntax as Lisp
    (mv-let
        (exc new-va)
```

```
        (interpret t
                   (safecdr (cdr stmt))
                   (declare-locals (safecar (cdr stmt)) va)
                   pa
                   clock)
      (mv exc
          (popn (len (safecar (cdr stmt))) new-va))))
   ((and (consp stmt)
         (equal (car stmt) 'proc-call))
    ;; (proc-call 'foo a b c d)
    (if (<= (fix-int clock) 0)
        (mv t va)
      (let ((proc (proc-lookup (safecar (cdr stmt)) pa))
            (actuals (safecdr (cdr stmt))))
        (mv-let (exc post-va)
            (interpret t
                       (proc-body proc)
                       (bind-formals (proc-formals proc)
                                     (eval-exp t actuals va))
                       pa
                       (- clock 1))
          (if exc
              (mv exc va)
            (mv nil (copy-out (proc-formals proc)
                              (proc-outlist proc)
                              actuals
                              post-va
                              va)))))))
   (t ; otherwise it's the null statement
    (mv nil va))))

;;; Finally, we prove lemmas for opening up calls of interpret.

(defthm nth-opener
  (implies (and (true-listp x)
                (integerp n)
                (<= 0 n))
           (equal (nth (+ 1 n) x)
                  (nth n (cdr x)))))

(defthm nth-1-interpret
  (equal (nth 1 (interpret flg stmt va pa clock))
         (cadr (interpret flg stmt va pa clock))))

;;; It's tempting to make the following lemma interpret-shape local to
;;; interpret-opener-cons, so that it doesn't clutter the database by
;;; hanging a rule (even a disabled one) under cons, but it may be
;;; useful sometime.  So let's leave it around, but of course let's
;;; leave it disabled once we're done with it.

(defthm interpret-shape
  (equal (list (car (interpret flg stmt va pa clock))
               (cadr (interpret flg stmt va pa clock)))
         (interpret flg stmt va pa clock)))

(defthm interpret-opener-cons
  (implies (equal mv1 (interpret nil stmt va pa clock))
           (equal (interpret t (cons stmt stmts) va pa clock)
                  (if (car mv1)
                      mv1
                    (interpret t stmts (cadr mv1) pa clock)))))

(in-theory (disable interpret-shape))

(defthm interpret-opener-nil
  (equal (interpret t nil va pa clock)
         (mv nil va)))

(defthm interpret-opener-assign
```

```
  (implies
   (and (consp stmt)
        (equal (car stmt) 'assign))
   (equal (interpret nil stmt va pa clock)
          (mv nil (variable-update
                   (safecar (cdr stmt))
                   (eval-exp nil (safecar (safecdr (cdr stmt))) va)
                   va)))))

(defthm interpret-opener-if
  (implies
   (and (consp stmt)
        (equal (car stmt) 'if))
   (equal (interpret nil stmt va pa clock)
          (if (equal 0 (eval-exp nil (safecar (cdr stmt)) va))
              (interpret nil (safecar (safecdr (safecdr (cdr stmt))))
                         va pa clock)
            (interpret nil (safecar (safecdr (cdr stmt)))
                       va pa clock)))))

(defthm interpret-opener-let*
  (implies
   (and (consp stmt)
        (equal (car stmt) 'let*))
   (equal (interpret nil stmt va pa clock)
          (mv-let (exc new-va)
                  (interpret t
                             (safecdr (cdr stmt))
                             (declare-locals (safecar (cdr stmt)) va)
                             pa
                             clock)
                  (list exc
                        (popn (len (safecar (cdr stmt)))
                              new-va))))))

;;; So much for the case of straight-line code.  Now let us prove
;;; lemmas for opening up the interpreter, carefully, on while loops.
;;; The following lemmas may appear to have the potential for being
;;; slightly inadequate, but in fact our approach is to use them only
;;; to prove the "generic" lemmas, after which we disable these three
;;; lemmas.

(defthm interpret-opener-while
  (implies (and (consp stmt)
                (equal (car stmt) 'while)
                (not (car (interpret nil stmt va pa clock))))
           (equal (interpret nil stmt va pa clock)
                  (if (equal (eval-exp nil (safecar (cdr stmt)) va) 0)
                      (mv nil va)
                    (interpret nil
                               stmt
                               (cadr (interpret t (safecdr (cdr stmt))
                                                 va pa clock))
                               pa
                               (+ -1 clock)))))
  ;; Let's avoid excessive opening up.
  :rule-classes ((:definition
                  :clique (interpret)
                  ;; May as well use the same controller as interpret
                  :controller-alist ((interpret nil t nil nil t))))
  :hints (("Goal" :expand (interpret nil stmt va pa clock))))

(defthm car-interpret-opener-while
  (implies
   (and (consp stmt)
        (equal (car stmt) 'while)
        (not (car (interpret nil stmt va pa clock)))
        (not (equal (eval-exp nil (safecar (cdr stmt)) va) 0))
        (equal new-va
```

```
               (cadr (interpret t (safecdr (cdr stmt)) va pa clock))))
   (not (car (interpret nil stmt new-va pa (+ -1 clock)))))
  :hints (("Goal" :expand (interpret nil stmt va pa clock)
           :in-theory (disable interpret))))

(defthm not-car-interpret-while-loop-to-body
  (implies (and (not (car (interpret nil stmt va pa clock)))
                (consp stmt)
                (equal (car stmt) 'while)
                (not (equal (eval-exp nil (safecar (cdr stmt)) va) 0))
                (equal body (safecdr (cdr stmt))))
           (not (car (interpret t body va pa clock))))
  :hints (("Goal" :expand (interpret nil stmt va pa clock))))

(defthm interpret-opener-proc-call
  (implies
   (and (consp stmt)
        (equal (car stmt) 'proc-call)
        (not (car (interpret nil stmt va pa clock))))
   (equal (interpret nil stmt va pa clock)
          (let ((proc (proc-lookup (safecar (cdr stmt)) pa))
                (actuals (safecdr (cdr stmt))))
            (mv nil
                (copy-out (proc-formals proc)
                          (proc-outlist proc)
                          actuals
                          (cadr (interpret t
                                           (proc-body proc)
                                           (bind-formals
                                            (proc-formals proc)
                                            (eval-exp t actuals va))
                                           pa
                                           (+ -1 clock)))
                          va)))))
  :hints (("Goal"
           :expand (interpret nil stmt va pa clock)))
  ;; Let's avoid excessive opening up.
  :rule-classes ((:definition
                  :clique (interpret)
                  ;; May as well use the same controller as interpret
                  :controller-alist ((interpret nil t nil nil t)))))

(defthm not-car-interpret-proc-call-to-body
  (implies
   (not (car (interpret nil
                        (list* 'proc-call
                               proc-name
                               proc-actuals)
                        va pa clock)))
   (not (car (interpret t (cdddr (proc-lookup proc-name pa))
                        (bind-formals (cadr (proc-lookup proc-name pa))
                                      (eval-exp t proc-actuals va))
                        pa (+ -1 clock)))))
  :hints (("Goal" :use
           ((:instance interpret-opener-proc-call
                       (stmt (list* 'proc-call
                                    proc-name proc-actuals))))
           :expand (interpret nil
                              (list* 'proc-call
                                     proc-name
                                     proc-actuals)
                              va pa clock)
           :in-theory (disable interpret-opener-proc-call
                               interpret))))

;; Now that we have proved opener lemmas for all the statement types,
;; we can safely disable interpret.

(in-theory (disable interpret))
```

Appendix B

generic

The file *generic.lisp* introduces the ‘‘generic’’ loop and procedure code and lemmas about them, for functional instantiation.

```
(in-package "ACL2")

(include-book "interpreter")

;;; Introduction of generic while loop and its various pieces and
;;; specificiations:

(encapsulate
 ((generic-while-expr () t)
  (generic-while-body () t)
  (generic-while-body-spec (va) t)
  (generic-while-invariant (va) t)
  (generic-while-measure (va) t)
  (generic-while-spec (va) t))

 (local
  (defun generic-while-expr () 0))

 (local
  (defun generic-while-body () nil))

 (local
  (defun generic-while-body-spec (va) va))

 (local
  (defun generic-while-body-spec-explicit (va) va))

 (local
  (defun generic-while-invariant (va) (declare (ignore va)) nil))

 (local
  (defun generic-while-measure (va) (declare (ignore va)) 0))

 (defthm e0-ordinalp-generic-while-measure
   (e0-ordinalp (generic-while-measure va)))

 (defthm generic-while-measure-decreases
   (implies
    (and (generic-while-invariant va)
         (not (equal 0 (eval-exp nil (generic-while-expr) va))))
    (e0-ord-< (generic-while-measure (generic-while-body-spec va))
              (generic-while-measure va))))

 (local
  (defun generic-while-spec (va)
    (if (equal (eval-exp nil (generic-while-expr) va) 0)
        va
      (generic-while-spec (generic-while-body-spec va)))))

 (defthm generic-while-spec-definition
   (implies
    (generic-while-invariant va)
    (equal (generic-while-spec va)
           (if (equal (eval-exp nil (generic-while-expr) va) 0)
               va
             (generic-while-spec (generic-while-body-spec va)))))
   :rule-classes
   ((:definition :clique (generic-while-spec)
                 :controller-alist ((generic-while-spec t)))))
```

```
 (defthm generic-while-body-spec-correct
   (implies
    (and (generic-while-invariant va)
         (not (equal 0 (eval-exp nil (generic-while-expr) va)))
         (not (car (interpret t (generic-while-body) va pa clock))))
    (equal (interpret t (generic-while-body) va pa clock)
           (list nil (generic-while-body-spec va)))))

 (defthm generic-while-invariant-preserved
   (implies
    (and (generic-while-invariant va)
         (not (equal 0 (eval-exp nil (generic-while-expr) va))))
    (generic-while-invariant (generic-while-body-spec va))))

 )

(defun generic-while-prog-correct-induction (va)
  (declare (xargs :measure (generic-while-measure va)))
   (if (or (not (generic-while-invariant va))
           (equal (eval-exp nil (generic-while-expr) va) 0))
       va
     (generic-while-prog-correct-induction
      (generic-while-body-spec va))))

(defun generic-while-prog ()
  (list* 'while (generic-while-expr) (generic-while-body)))

(in-theory (disable (generic-while-prog)))

(defthm generic-while-prog-correct-lemma
  (implies (and (generic-while-invariant va)
                (equal program (generic-while-prog))
                (not (car (interpret nil program va pa clock))))
           (equal (interpret nil program va pa clock)
                  (list nil (generic-while-spec va))))
  :hints (("Goal" :induct (interpret nil program va pa clock)
           :expand (interpret nil
                              (list* 'while
                                     (generic-while-expr)
                                     (generic-while-body))
                              va pa clock)))
  :rule-classes nil)

(defthm generic-while-prog-correct
  (implies (and (generic-while-invariant va)
                (not (car (interpret nil (generic-while-prog)
                                     va pa clock))))
           (equal (interpret nil (generic-while-prog) va pa clock)
                  (list nil (generic-while-spec va))))
  :hints (("Goal" :use
           (:instance
            generic-while-prog-correct-lemma
            (program (generic-while-prog))))))

(in-theory (disable not-car-interpret-while-loop-to-body
                    interpret-opener-while
                    car-interpret-opener-while))

;;; Now let us introduce the explicit versions of the generic
;;; specifications.

(encapsulate
 ((generic-while-body-spec-explicit (va) t))

 (local
  (defun generic-while-body-spec-explicit (va)
    (generic-while-body-spec va)))

 (defthm generic-while-body-spec-is-generic-while-body-spec-explicit
```

```
   (implies (and (not (equal (eval-exp nil (generic-while-expr) va)
                             0))
                 (generic-while-invariant va))
            (equal (generic-while-body-spec va)
                   (generic-while-body-spec-explicit va)))))

(encapsulate
 ((generic-while-spec-explicit (va) t))

 (local
  (defun generic-while-spec-explicit (va)
    (generic-while-spec va)))

 (defthm generic-while-spec-explicit-correct-base
   (implies (and (equal (eval-exp nil (generic-while-expr) va)
                        0)
                 (generic-while-invariant va))
            (equal (generic-while-spec-explicit va) va)))

 (defthm generic-while-spec-explicit-correct-step
   (implies
    (and (not (equal (eval-exp nil (generic-while-expr) va)
                     0))
         (generic-while-invariant va))
    (equal (generic-while-spec-explicit
            (generic-while-body-spec-explicit va))
           (generic-while-spec-explicit va)))))

(defthm generic-while-invariant-preserved-explicit-case
  (implies
   (and (generic-while-invariant va)
        (not (equal 0 (eval-exp nil (generic-while-expr) va))))
   (generic-while-invariant (generic-while-body-spec-explicit va)))
  :hints (("Goal" :use generic-while-invariant-preserved
           :in-theory (disable generic-while-invariant-preserved))))

(defthm generic-while-spec-explicit-correct
  (implies (generic-while-invariant va)
           (equal (generic-while-spec va)
                  (generic-while-spec-explicit va)))
  :hints (("Goal" :induct (generic-while-prog-correct-induction va))))

(defthm generic-while-prog-explicit-correct
  (implies
   (and (generic-while-invariant va)
        (not (car (interpret nil (generic-while-prog) va pa clock))))
   (equal (interpret nil (generic-while-prog) va pa clock)
          (list nil (generic-while-spec-explicit va))))
  :hints (("Goal" :in-theory
           (union-theories '(generic-while-spec-explicit-correct
                             generic-while-prog-correct)
                           *s-prop-theory*)))
  :rule-classes nil)

(in-theory (disable generic-while-spec-explicit-correct
                    generic-while-prog-correct))

;;; Finally let's handle proc calls generically.  In practice, we'll
;;; probably explicitly allow non-recursive procedure calls to be
;;; opened up rather directly (using interpret-opener-proc-call), but
;;; for recursive procedures we'll have to engage in something
;;; analogous to the while loop case.

;;; Let's develop the nonrecursive case first.  Notice that inside a
;;; procedure body, we know exactly what the variable alist looks
;;; like.  So, we may as well take advantage of that information.
;;; Hence, our proof obligation is simply to show that the body spec
;;; is satisfied by an alist of the expected shape.
```

```
(encapsulate
 ((generic-proc-name () t)
  (generic-proc-actuals () t)
  (generic-proc-formals () t)
  (generic-proc-outlist () t)
  (generic-proc-body () t)
  (generic-proc-entry (vals) t)
  (generic-proc-body-spec (va) t)
  (generic-proc-spec (va) t))

 (local
  (defun generic-proc-name () 'proc))

 (local
  (defun generic-proc-actuals () nil))

 (local
  (defun generic-proc-formals () nil))

 (defthm generic-proc-formals-true-listp
   (true-listp (generic-proc-formals))
   :rule-classes :type-prescription)

 (local
  (defun generic-proc-outlist () nil))

 (defthm generic-proc-outlist-true-listp
   (true-listp (generic-proc-outlist))
   :rule-classes :type-prescription)

 (local
  (defun generic-proc-body () nil))

 (defthm generic-proc-body-true-listp
   (true-listp (generic-proc-body))
   :rule-classes :type-prescription)

 (local
  (defun generic-proc-entry (vals) (declare (ignore vals)) t))

 (local
  (defun generic-proc-body-spec (va) va))

 (local
  (defun generic-proc-spec (va) va))

 ;; Notice that when we functionally-instantiate the following, we
 ;; have the option of ignoring the first hypothesis.

 (defthm generic-proc-body-spec-correct
   (implies (and (generic-proc-entry vals)
                 (equal body-va
                        (bind-formals (generic-proc-formals)
                                      vals))
                 (not (car (interpret t
                                      (generic-proc-body)
                                      body-va
                                      pa clock))))
            (equal (interpret t (generic-proc-body) body-va pa clock)
                   (list nil (generic-proc-body-spec body-va)))))

#| The following would suffice, but hopefully the stronger lemma
   below will not be too strong, since it's simpler.
 (implies
  (and (equal (proc-lookup (generic-proc-name) pa)
              (list* (generic-proc-name)
                     (generic-proc-formals)
                     (generic-proc-outlist)
                     (generic-proc-body)))
```

```
       (not (car (interpret nil
                            (list* 'proc-call
                                   (generic-proc-name)
                                   (generic-proc-actuals))
                            va pa clock))))
  (equal (generic-proc-spec va)
         (copy-out (generic-proc-formals)
                   (generic-proc-outlist)
                   (generic-proc-actuals)
                   (generic-proc-body-spec
                    (bind-formals
                     (generic-proc-formals)
                     (eval-exp t (generic-proc-actuals) va)))
                   va)))
|#

 (defthm generic-proc-spec-correct
   (implies (and (equal vals (eval-exp t (generic-proc-actuals) va))
                 (generic-proc-entry vals))
            (equal (generic-proc-spec va)
                   (copy-out (generic-proc-formals)
                             (generic-proc-outlist)
                             (generic-proc-actuals)
                             (generic-proc-body-spec
                              (bind-formals (generic-proc-formals)
                                            vals))
                             va))))
 )

(defun generic-proc ()
  (list* (generic-proc-name)
         (generic-proc-formals)
         (generic-proc-outlist)
         (generic-proc-body)))

(defun generic-proc-call ()
  (list* 'proc-call (generic-proc-name) (generic-proc-actuals)))

(in-theory (disable (generic-proc-call) (generic-proc)))

(encapsulate
 ()

 (local
  (defthm not-car-interpret-proc-call-to-body-better-lemma-1
    (equal (EQUAL x
                  (LIST* PROC-NAME
                         PROC-FORMALS PROC-OUTLIST PROC-BODY))
           (and (consp x)
                (consp (cdr x))
                (consp (cddr x))
                (equal (car x)
                       proc-name)
                (equal (cadr x)
                       proc-formals)
                (equal (caddr x)
                       proc-outlist)
                (equal (cdddr x)
                       proc-body)))))

 (local
  (defthm not-car-interpret-proc-call-to-body-better-lemma-2
    (equal (equal (proc-lookup proc-name pa)
                  (list* proc-name
                         proc-formals proc-outlist proc-body))
           (and (consp (proc-lookup proc-name pa))
                (consp (cdr (proc-lookup proc-name pa)))
                (consp (cddr (proc-lookup proc-name pa)))
                (equal (car (proc-lookup proc-name pa))
```

```
                       proc-name)
                (equal (cadr (proc-lookup proc-name pa))
                       proc-formals)
                (equal (caddr (proc-lookup proc-name pa))
                       proc-outlist)
                (equal (cdddr (proc-lookup proc-name pa))
                       proc-body)))))

 (defthm not-car-interpret-proc-call-to-body-better
   (implies
    (and (equal (proc-lookup proc-name pa)
                (list* proc-name
                       proc-formals
                       proc-outlist
                       proc-body))
         (not (car (interpret nil
                              (list* 'proc-call
                                     proc-name
                                     proc-actuals)
                              va pa clock))))
    (not (car (interpret t proc-body
                         (bind-formals proc-formals
                                       (eval-exp t proc-actuals va))
                         pa (+ -1 clock)))))
   :hints (("Goal" :use not-car-interpret-proc-call-to-body
            :in-theory (disable not-car-interpret-proc-call-to-body))))
 )

(defthm generic-proc-call-correct
  (implies
   (and (generic-proc-entry (eval-exp t (generic-proc-actuals) va))
        (equal (proc-lookup (generic-proc-name) pa)
               (generic-proc))
        (not (car (interpret nil (generic-proc-call) va pa clock))))
   (equal (interpret nil (generic-proc-call) va pa clock)
          (list nil (generic-proc-spec va)))))

;;; Let us prove some lemmas about fix-int so that we can disable it.

;;; The following lemma causes fix-int to open and introduces the
;;; integerp question so that the other lemmas can fire.  The idea is
;;; that it fires when we consider termination of while loops.

(defthm equal-fix-int-0
  (equal (equal 0 (fix-int x))
         (or (equal 0 x)
             (not (integerp x)))))

(defthm fix-int-not-integerp
  (implies (not (integerp x))
           (equal (fix-int x) 0)))

(defthm fix-int-integerp
  (implies (integerp x)
           (equal (fix-int x) x)))

(in-theory (disable fix-int))

;;; In this approach, when we need forcing we enable it.

(disable-forcing)

;;; A convenient macro:

(defmacro fix-nat (x)
  `(max 0 (fix-int ,x)))
```

Appendix C
force

The file *force.lisp* defines a theory containing lemmas with **`force`**d hypotheses, for use in a symbolic evaluator defined in *macros.lisp*.

```
;;; This file contains lemmas that have some connection with FORCing.
;;; Some of them could reasonably be included elsewheere, no doubt.
;;; The main lemmas are interpret-opener-cons-force and
;;; interpret-opener-let*-force, which force the rewriter to disallow
;;; the possibility of exceptions in let* statements and in the first
;;; statement in a list of statements.  This is useful for the
;;; symbolic evaluator, which could otherwise be tempted to a wimpy in
;;; opening up such interpreter calls.

(in-package "ACL2")

(include-book "interpreter")

(deflabel begin-force-theory)

(defthm fix-int-integerp-force
  (implies (force (integerp x))
           (equal (fix-int x) x)))

(defthm nth-opener-force
  (implies (and (force (true-listp x))
                (force (integerp n))
                (force (<= 0 n)))
           (equal (nth (+ 1 n) x)
                  (nth n (cdr x)))))

(defthm interpret-opener-cons-force
  (implies (and (equal mv1 (interpret nil stmt va pa clock))
                (force (not (car mv1))))
           (equal (interpret t (cons stmt stmts) va pa clock)
                  (interpret t stmts (cadr mv1) pa clock))))

(defthm interpret-opener-let*-force
  (implies (and (consp stmt)
                (equal (car stmt) 'let*)
                (equal mv (interpret
                           t
                           (safecdr (cdr stmt))
                           (declare-locals (safecar (cdr stmt)) va)
                           pa
                           clock))
                (force (not (car mv))))
           (equal (interpret nil stmt va pa clock)
                  (list nil
                        (popn (len (safecar (cdr stmt)))
                              (cadr mv))))))

(defthm variable-value-variable-update-force
  (implies (force (variable-boundp x va))
           (equal (variable-value x (variable-update x val va))
                  val)))

;;; The following deflabel isn't necessary -- the current-theory hint
;;; in force-theory could instead refer to :here -- but this seems
;;; good form.

(deflabel end-force-theory)

(deftheory force-theory
```

```
  (set-difference-theories
   (union-theories '((force))
                   (current-theory 'end-force-theory))
   (current-theory 'begin-force-theory)))

(in-theory (disable force-theory))
```

Appendix D

macros

The file *macros.lisp* provides a definition of our analogue of a VCG, **`defprogram`**.

```
(in-package "ACL2")

(include-book "interpreter")
(include-book "generic")
(include-book "force")

(red)

(table abb-table
       nil nil
       :guard
       (and (symbolp val)
            (if (legal-constantp val)
                (let ((term (getprop val 'const nil
                                     'current-acl2-world world)))
                  (and term
                       (equal term key)))
              (let ((x (symbol-name val)))
                (and (not (equal x ""))
                     (eql (char x 0) #\&))))))

(defmacro defc (name val &rest args)
  `(progn (defconst ,name ,val ,@args)
          (table abb-table
                 (getprop ',name 'const
                          '(:error "DEFC failed")
                          'current-acl2-world world)
                 ',name)))

(defun untranslate-abb1 (term abbs iff-flg)
  (untranslate (if abbs
                   (sublis-expr-non-quoteps abbs term)
                 term)
               iff-flg))

(defun untranslate-abb (term wrld iff-flg)
  (untranslate-abb1 term (table-alist 'abb-table wrld) iff-flg))

(defun prettyify-clause1-abb (cl abbs)
  (cond ((null (cdr cl)) nil)
        (t (cons (dumb-negate-lit (untranslate-abb1 (car cl) abbs t))
                 (prettyify-clause1-abb (cdr cl) abbs)))))

(defun prettyify-clause-abb (cl abbs)
  (cond ((null cl) nil)
        ((null (cdr cl))
         (untranslate-abb1 (car cl) abbs t))
        ((null (cddr cl))
         (list 'implies
               (dumb-negate-lit (untranslate-abb1 (car cl) abbs t))
               (untranslate-abb1 (cadr cl) abbs t)))
        (t (list 'implies
                 (cons 'and (prettyify-clause1-abb cl abbs))
                 (untranslate-abb1 (car (last cl)) abbs t)))))

(defun prettyify-clause-lst-abb (clauses abbs)
  (cond ((null clauses) nil)
        (t (cons (prettyify-clause-abb (car clauses) abbs)
                 (prettyify-clause-lst-abb (cdr clauses) abbs)))))
```

```
(defun display-constraints (lmi fnl-substn raw-hints wrld state)
  (state-global-let*
   ((inhibit-output-lst '(prove summary)))
   (er-let*
    ((pair (translate-lmi (list* :functional-instance lmi fnl-substn)
                          (cons 'display-constraints lmi)
                          wrld
                          state)))
    (let ((constraints (conjoin (cdr pair))))
      (er-let*
       ((hints (translate-hints
                'display-constraints
                `(("Goal" :do-not
                   '(generalize eliminate-destructors
                                fertilize eliminate-irrelevance)
                   :do-not-induct display-constraints
                   ,@raw-hints))
                (cons 'display-constraints '|some lmi|)
                wrld
                state))
        (ttree (pc-prove constraints
                         (untranslate constraints t)
                         hints
                         t
                         wrld
                         (cons 'display-constraints '|some lmi|)
                         state)))
       (value (prettyify-clause-lst-abb
               (unproved-pc-prove-terms ttree)
               (table-alist 'abb-table wrld)))))))))

(defun create-subsidiary-defthms (name index thms)
  (if thms
      (cons (list
             'defthm
             (intern-in-package-of-symbol
              (coerce (append (coerce (symbol-name name) 'list)
                              (cons #\- (explode-nonnegative-integer
                                         index nil)))
                      'string)
              name)
             (car thms))
            (create-subsidiary-defthms name (1+ index) (cdr thms)))
    nil))

(defmacro defthm-expand (name thm &key hints)
  (declare (ignore thm)) ;later I may add the subsumption check
  (case-match
   hints
   ((('"Goal" ':by
      (':functional-instance
       previous-name . fnl-subst) . raw-hints))
    `(er-let*
      ((constraints
        (display-constraints ',previous-name ',fnl-subst ',raw-hints
                             (w state) state)))
      (value (create-subsidiary-defthms ',name 1 constraints))))
   (& ''(wrong form for defthm-expand))))

(defun declare-fn (vars)
  (declare (xargs :guard (true-listp vars)))
  (if vars
      (cons `(variable-boundp ',(car vars) va)
            (cons `(integerp (variable-value ',(car vars) va))
                  (declare-fn (cdr vars))))
    nil))

(defmacro @declare (&rest vars)
  (cons 'and (declare-fn vars)))
```

```
(defun suffix (sym suf)
  (declare (xargs :color :red))
  (intern-in-package-of-symbol
   (string-append (symbol-name sym)
                  suf)
   sym))

(defun prefix (sym prf)
  (declare (xargs :color :red))
  (intern-in-package-of-symbol
   (string-append prf
                  (symbol-name sym))
   sym))

(defun make-const (name suf)
  (prefix
   (suffix name (string-append suf "*"))
   "*"))

(defun auxiliary-name-doublets
  (root-name const-suffixes other-suffixes)
  (declare (xargs :guard
                  (and (symbolp-listp const-suffixes)
                       (symbolp-listp other-suffixes)
                       (not (intersectp-eq const-suffixes
                                           other-suffixes)))))
  (if const-suffixes
      (cons (list (car const-suffixes)
                  `(make-const ,root-name
                               ,(symbol-name (car const-suffixes))))
            (auxiliary-name-doublets root-name
                                     (cdr const-suffixes)
                                     other-suffixes))
    (if other-suffixes
        (cons (list (car other-suffixes)
                    `(suffix ,root-name
                             ,(symbol-name (car other-suffixes))))
              (auxiliary-name-doublets root-name
                                       const-suffixes
                                       (cdr other-suffixes)))
      nil)))

(defmacro defprogram-fn-guts
  (root-name const-suffixes other-suffixes form)
  `(let ,(auxiliary-name-doublets
          root-name
          const-suffixes
          other-suffixes)
     ,form))

(defun initial-va (vars)
  (if vars
      (cons (cons (car vars)
                  (list '@pre (car vars)))
            (initial-va (cdr vars)))
    nil))

(defun merge-term-order-no-dups (l1 l2)
  ;; adapted from merge-term-order of acl2
  (cond ((null l1) l2)
        ((null l2) l1)
        ((equal (car l1) (car l2))
         (cons (car l1) (merge-term-order-no-dups (cdr l1) (cdr l2))))
        ((term-order (car l1) (car l2))
         (cons (car l1) (merge-term-order-no-dups (cdr l1) l2)))
        (t (cons (car l2) (merge-term-order-no-dups l1 (cdr l2))))))

(defun exp-vars (list-flag x)
  (if list-flag
```

```
      (if (consp x)
          (merge-term-order-no-dups (exp-vars nil (car x))
                                    (exp-vars t (cdr x)))
        nil)
    (cond ((integerp x) nil)
          ((atom x)
           (list x))
          ((or (eq (car x) '+)
               (eq (car x) 'plus)
               (eq (car x) '*)
               (eq (car x) 'times)
               (eq (car x) 'lessp))
           (merge-term-order-no-dups
            (exp-vars nil (safecar (cdr x)))
            (exp-vars nil (safecar (safecdr (cdr x))))))
          ((or (eq (car x) '-)
               (eq (car x) 'negative))
           (exp-vars nil (safecar (cdr x))))
          (t (er hard 'exp-vars "Undefined expression, ~p0." x)))))

(defun program-vars (flg stmt)
  (cond
   (flg
    (if (consp stmt)
        (merge-term-order-no-dups (program-vars nil (car stmt))
                                  (program-vars t (cdr stmt)))
      nil))
   ((and (consp stmt)
         (equal (car stmt) 'assign))
    (merge-term-order-no-dups
     (list (safecar (cdr stmt)))
     (exp-vars nil (safecar (safecdr (cdr stmt))))))
   ((and (consp stmt)
         (member-eq (car stmt) '(if while)))
    (merge-term-order-no-dups (exp-vars nil (safecar (cdr stmt)))
                              (program-vars t (safecdr (cdr stmt)))))
   ((and (consp stmt)
         (equal (car stmt) 'let*))
    (set-difference-equal (program-vars t (safecdr (cdr stmt)))
                          (strip-cars (safecar (cdr stmt)))))
   ((and (true-listp stmt)
         (equal (car stmt) 'proc-call))
    (exp-vars t (safecdr (cdr stmt))))
   (t
    (er hard 'program-vars "Undefined statement, ~p0." stmt))))

(defun force-theory-ens (state)
  (let ((world (w state)))
    (load-theory-into-enabled-structure
     (union-theories (theory 'force-theory)
                     (current-theory :here))
     (ens state)
     t
     world)))

(defconst *max-repeats-for-fast-rewrite* 5)

(defun symbolic-eval (term state)
  (let ((wrld (w state))
        (ens (force-theory-ens state)))
    (mv-let (new-term ttree state)
        (pc-rewrite* term nil nil nil wrld
                     (change rewrite-constant
                             (initial-rcnst wrld)
                             :current-enabled-structure
                             ens)
                     nil nil
                     t ;normalize-flg
                     t ;rewrite-flg
```

```
                     ens
                     state
                     *max-repeats-for-fast-rewrite*)
      (declare (ignore ttree))
      (value new-term))))

(mutual-recursion

(defun untranslate-variable-value (term)
  (cond
   ((or (variablep term)
        (fquotep term))
    term)
   ((and (eq (ffn-symb term) 'variable-value)
         (quotep (fargn term 1))
         (eq (fargn term 2) 'va))
    `(@pre ,(cadr (fargn term 1))))
   (t (fcons-term (ffn-symb term)
                  (untranslate-variable-value-lst (fargs term))))))

(defun untranslate-variable-value-lst (lst)
  (if lst
      (cons (untranslate-variable-value (car lst))
            (untranslate-variable-value-lst (cdr lst)))
    nil))

)

(defun variable-update-term-to-alist (term)
  (cond
   ((eq term 'va)
    (mv nil nil))
   ((or (variablep term)
        (fquotep term)
        (not (eq (ffn-symb term) 'variable-update))
        (not (quotep (fargn term 1))))
    (mv t term))
   (t
    (mv-let (erp val)
        (variable-update-term-to-alist (fargn term 3))
      (if erp
          (mv erp val)
        (mv nil
            (cons (cons (cadr (fargn term 1)) (fargn term 2))
                  val)))))))

(defun merge-variable-update-alists (test vars alist1 alist2)
  (if vars
      (cons
       (cons
        (car vars)
        (fcons-term*
         'if ;*** consider 'if* here
         test
         (or (cdr (assoc-eq (car vars) alist1))
             (fcons-term* 'variable-value (kwote (car vars)) 'va))
         (or (cdr (assoc-eq (car vars) alist2))
             (fcons-term* 'variable-value (kwote (car vars)) 'va))))
       (merge-variable-update-alists test (cdr vars) alist1 alist2))
    nil))

(defun untranslate-variable-update-mv1 (vars term)
  (if (and (nvariablep term)
           (not (fquotep term)))
      (if (and (eq (ffn-symb term) 'cons)
               (equal (fargn term 1) *nil*)
               (let ((x (fargn term 2)))
                 (and (nvariablep x)
                      (not (fquotep x))
```

```
                      (eq (ffn-symb x) 'cons)
                      (equal (fargn x 2) *nil*))))
          (variable-update-term-to-alist (fargn (fargn term 2) 1))
        (if (eq (ffn-symb term) 'if)
            (mv-let (exc1 val1)
                (untranslate-variable-update-mv1 vars (fargn term 2))
              (if exc1
                  (mv exc1 val1)
                (mv-let (exc2 val2)
                    (untranslate-variable-update-mv1
                     vars (fargn term 3))
                  (if exc2
                      (mv exc2 val2)
                    (mv nil
                        (merge-variable-update-alists
                         (fargn term 1)
                         vars
                         val1
                         val2))))))
          (mv t term)))
    (mv t term)))

(defun simplify-untranslate-variable-update-mv1 (alist state)
  ;; here we finally build doublets
  (if alist
      (er-let* ((x (symbolic-eval (cdar alist) state))
                (rst (simplify-untranslate-variable-update-mv1
                      (cdr alist) state)))
               (value
                (cons
                 (list (caar alist)
                       (untranslate (untranslate-variable-value x)
                                    nil))
                 rst)))
    (value nil)))

(defun untranslate-variable-update-mv (vars term state)
  (mv-let (erp val)
      (untranslate-variable-update-mv1 vars term)
    (if erp
        (mv erp val state)
      (simplify-untranslate-variable-update-mv1 val state))))

(defun make-body-spec (code state)
  ;; code is of the form (quote (while test . body))
  (if (and (consp code)
           (eq (car code) 'quote)
           (consp (cadr code))
           (eq (car (cadr code)) 'while))
      (let ((body-code (cddr (cadr code))))
        (er-let* ((term (symbolic-eval `(interpret 't
                                                  ',body-code
                                                  va
                                                  'nil
                                                  '100)
                                       state)))
                 (mv-let (erp val state)
                     (untranslate-variable-update-mv
                      (program-vars t body-code) term state)
                   (if erp
                       (mv "Was unable to simplify the execution of ~
                            the body of the while loop,~%  ~p0,~%to ~
                            an @passign expression.  The ~
                            unsimplified expression was:~%  ~p1~%"
                           (list (cons #\0 body-code)
                                 (cons #\1 term))
                           state)
                     (value (cons '@passign val))))))
    (mv "Expected an explicit while loop, but got:~%  ~p0.~%"
```

```
        (list (cons #\0 code))
        state)))

(defun defprogram-fn
  (name code invariant exit body-spec measure initial-commands state)
  (let ((program-name (make-const name "")))
    (mv-let
     (erp body-spec state)
     (if body-spec
         (value body-spec)
         (make-body-spec code state))
     (if erp
         (mv erp body-spec state)
         (value
          (defprogram-fn-guts
            name
            ($body $test)
            ($start
             $invariant $body-spec-guts
             $body-spec $body-correct
             $measure $e0-ordinalp$measure $measure-decreases
             $spec $invariant-preserved $correct
             $spec-explicit-guts $spec-explicit
             $invariant-preserved-guts $spec-explicit-correct-1
             $spec-explicit-correct-2 $spec-explicit-correct-3
             $spec-explicit-correct-base $spec-explicit-correct-step
             $spec-explicit-correct $correct-explicit)
            (let ((measure (or measure
                                   `(fix-nat (eval-exp nil ,$test va)))))
              `((in-package "ACL2")
                (deflabel ,$start)
                ,@initial-commands
                (defc ,program-name ,code)
                (defc ,$body (cddr ,program-name))
                (defconst ,$test (cadr ,program-name))
                (defun ,$invariant (va) ,invariant)
                (defabbrev ,$body-spec-guts (va) ,body-spec)
                (defun ,$body-spec (va)
                  (declare (xargs :guard (,$invariant va)))
                  (,$body-spec-guts va))
                (local
                 (deflabel ,(suffix name "-COMMENT-1")
                   :doc
                   "The following should proceed automatically, but ~
                    is specific to the current program."))
                (defthm ,$body-correct
                  (implies
                   (and (not (car (interpret t ,$body va pa clock)))
                        (not (equal 0 (eval-exp nil ,$test va)))
                        (,$invariant va))
                   (equal (interpret t ,$body va pa clock)
                          (list nil (,$body-spec va)))))
                (defun ,$measure (va)
                  (declare (xargs :color :blue))
                  ,measure)
                (defthm ,$e0-ordinalp$measure
                  (e0-ordinalp (,$measure va)))
                (defthm ,$measure-decreases
                  (implies
                   (and (,$invariant va)
                        (not (equal 0 (eval-exp nil ,$test va))))
                   (e0-ord-< (,$measure (,$body-spec va))
                             (,$measure va))))
                (defun ,$spec (va)
                  (declare (xargs :measure (,$measure va)))
                  (if (or (not (,$invariant va))
                          (equal (eval-exp nil ,$test va) 0))
                      va
                      (,$spec (,$body-spec va))))
```

```
(local
 (deflabel ,(suffix name "-COMMENT-2")
   :doc
   "The following is specific to the current ~
    program, but typically is not too difficult."))
(defthm ,$invariant-preserved
  (implies
   (and (,$invariant va)
        (not (equal 0 (eval-exp nil ,$test va))))
   (,$invariant (,$body-spec va))))
(defthm ,$correct
  (implies (and (,$invariant va)
                (not (car (interpret nil
                                     ,program-name
                                     va pa clock))))
           (equal (interpret nil ,program-name
                             va pa clock)
                  (list nil (,$spec va))))
  :hints (("Goal"
           :by
           (:functional-instance
            generic-while-prog-correct
            (generic-while-measure ,$measure)
            (generic-while-prog
             (lambda () ,program-name))
            (generic-while-body (lambda () ,$body))
            (generic-while-spec ,$spec)
            (generic-while-body-spec ,$body-spec)
            (generic-while-expr (lambda () ,$test))
            (generic-while-invariant ,$invariant))
           :in-theory (union-theories
                       '(,$e0-ordinalp$measure
                         ,$measure-decreases
                         ,$spec
                         ,$body-correct
                         ,$invariant-preserved)
                       *s-prop-theory*))))
(defabbrev ,$spec-explicit-guts (va)
  ,exit)
(defun ,$spec-explicit (va)
  (declare (xargs :guard (,$invariant va)))
  (,$spec-explicit-guts va))
(defthm ,$invariant-preserved-guts
  (implies
   (and (,$invariant va)
        (not (equal 0 (eval-exp nil ,$test va))))
   (,$invariant (,$body-spec-guts va)))
  :hints (("Goal" :use ,$invariant-preserved
           :expand ((,$body-spec va))
           :in-theory *s-prop-theory*)))
(encapsulate
 ()
 (local
  (defthm ,$spec-explicit-correct-1
    (implies (and (,$invariant va)
                  (equal (eval-exp nil ,$test va) 0))
             (equal (,$spec va) va))
    :hints (("Goal" :expand ((,$spec va))
             :in-theory *s-prop-theory*))))
 (local
  (defthm ,$spec-explicit-correct-2
    (implies
     (and (,$invariant va)
          (not (equal (eval-exp nil ,$test va) 0)))
     (equal (,$spec (,$body-spec va))
            (,$spec va)))
    :hints (("Goal" :expand ((,$spec va))
             :in-theory *s-prop-theory*))))
 (local
```

```
 (defthm ,$spec-explicit-correct-3
   (implies
    (and (not (equal (eval-exp nil ,$test va) 0))
         (,$invariant va)
         (equal va2 (,$body-spec-guts va)))
    (equal (equal
            (,$body-spec va)
            va2)
           t))
   :hints (("Goal" :expand ((,$body-spec va))
            :in-theory *s-prop-theory*))))
(local
 (deflabel ,(suffix name "-COMMENT-3")
   :doc
   "The following is specific to the current ~
    program, but probably should be very easy."))
(local
 (defthm ,$spec-explicit-correct-base
   (implies (and (equal (eval-exp nil ,$test va) 0)
                 (,$invariant va))
            (equal (,$spec-explicit va)
                   va))))
(local
 (deflabel ,(suffix name "-COMMENT-4")
   :doc
   "The following is specific to the current ~
    program, and is really the heart of the ~
    matter."))
(local
 (defthm ,$spec-explicit-correct-step
   (implies
    (and (not (equal (eval-exp nil ,$test va) 0))
         (,$invariant va)
         (equal va2 (,$body-spec-guts va)))
    (equal (equal
            (,$spec-explicit va2)
            (,$spec-explicit va))
           t))))
(defthm ,$spec-explicit-correct
  (implies (,$invariant va)
           (equal (,$spec va)
                  (,$spec-explicit va)))
  :hints (("Goal" :by
           (:functional-instance
            generic-while-spec-explicit-correct
            (generic-while-measure ,$measure)
            (generic-while-body (lambda () ,$body))
            (generic-while-spec ,$spec)
            (generic-while-body-spec ,$body-spec)
            (generic-while-expr (lambda () ,$test))
            (generic-while-invariant ,$invariant)
            (generic-while-spec-explicit
             (lambda (va) (,$spec-explicit va)))
            (generic-while-body-spec-explicit
             (lambda (va) (,$body-spec-guts va))))
           :in-theory
           (union-theories
            '(,$e0-ordinalp$measure
              ,$measure-decreases
              ,$body-correct
              ,$invariant-preserved-guts
              ,$invariant-preserved
              ,$spec-explicit-correct-1
              ,$spec-explicit-correct-2
              ,$spec-explicit-correct-3
              ,$spec-explicit-correct-base
              ,$spec-explicit-correct-step)
            *s-prop-theory*))))
)
```

```
                (defthm ,$correct-explicit
                  (implies
                   (and (force (,$invariant va))
                        (force (not (car (interpret nil ,program-name
                                                    va pa clock)))))
                   (equal (interpret nil ,program-name va pa clock)
                          (list nil (,$spec-explicit va))))
                  :hints (("Goal" :in-theory
                           (union-theories '(,$correct
                                             ,$spec-explicit-correct
                                             force)
                                           *s-prop-theory*)))))))))))

(defun find-new-filename (root-name index overwrite state)
  (let ((filename (if (equal index 0)
                      (string-append root-name ".lisp")
                    (string-append
                     root-name
                     (string-append
                      (coerce (cons #\.
                                    (explode-nonnegative-integer
                                     index nil))
                              'string)
                      ".lisp")))))
    (if overwrite
        (mv filename state)
      (mv-let (chan state)
          (open-input-channel filename :object state)
        (if chan
            (find-new-filename root-name (1+ index) overwrite state)
          (mv filename state))))))

(defun print-objects-pretty (lst chan state)
  (if (null lst)
      state
    (pprogn (fms "~q0" (list (cons #\0 (car lst))) chan state nil)
            (print-objects-pretty (cdr lst) chan state))))

(defun fix-invariant (term vars)
  `(and (@declare ,@vars)
        ,term))

(defmacro defprogram
  (&whole form name body &key invariant exit body-spec filename
          measure initial-commands (overwrite 't))
  `(mv-let (erp lst state)
       (defprogram-fn
         ',name
         (list 'quote ,body)
         (fix-invariant ',invariant (program-vars nil ,body))
         ',exit ',body-spec ',measure ',initial-commands state)
     (if erp
         (pprogn
          (if (stringp erp)
              (fms erp lst (standard-co state) state nil)
            (fms "~%Error in (defprogram ~p0 ...).~%"
                 (list (cons #\0 ',name))
                 (standard-co state) state nil))
          (mv t nil state))
       (mv-let (filename state)
           ,(if filename
                `(mv ',filename state)
              `(find-new-filename
                ,(string-downcase (symbol-name name))
                0
                ',overwrite
                state))
         (mv-let (chan state)
             (open-output-channel filename :object state)
```

```
           (pprogn
            (fms "Writing events to file ~p0.~%"
                 (list (cons #\0 filename))
                 *standard-co*
                 state
                 nil)
            (fms "#| Created from:~%~p0~%|#~%"
                 (list (cons #\0 ',form))
                 chan state nil)
            (print-objects-pretty lst chan state)
            (value :invisible)))))))

(defmacro @pre (var &optional (va 'va))
  `(variable-value ',var ,va))

(defun @passign-fn (pairs)
  (declare (xargs :color :red))
  (if pairs
      `(variable-update
        ',(caar pairs)
        ,(cadar pairs)
        ,(@passign-fn (cdr pairs)))
    'va))

(defmacro @passign (&rest pairs)
  ;;(@passign (x (+ (@pre x) (@pre y))) (y (- (@pre x))))
  (@passign-fn pairs))
```

References

1. William D. Young, ‘‘A Mechanically Verified Code Generator’’, *Journal of Automated Reasoning*, Vol. 5, No. 4, December 1989, pp. 493-518, Also published as CLI Technical Report 30

2. R.S. Boyer, D. Goldschlag, M. Kaufmann, J S. Moore, ‘‘Functional Instantiation in First Order Logic’’, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, 1991, pp. 7-26, Versions also published as CLI Technical Report 44 and in proceedings of the 1989 Workshop on Programming Logic, Programming Methodology Group, University of Goteborg, West Germany

3. R.S. Boyer, J S. Moore, ‘‘A Theorem Prover for a Computational Logic’’, Technical Report 54, Computational Logic, Inc., 1990, Published in proceedings of 10th International Conference on Automated Deduction, Kaiserslautern, West Germany, July 1990

Table of Contents