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

```
;   The following list of events includes every theorem
;   claimed to have been proved in the paper

;               The Addition of Bounded Quantification
;                                 and
;                          Partial Functions
;                                  to
;                        A Computational Logic
;                                 and
;                          Its Theorem Prover

;   except for the Binomial Theorem, which is part of the
;   basic proveall.
```

`; Section 1.  Introduction`

DEFINITION:
```
double-list (l)
=   if l ≃ nil then nil
    else cons (2 * car (l), double-list (cdr (l))) endif
```

THEOREM: double-list-append
```
double-list (append (a, b)) = append (double-list (a), double-list (b))
```

THEOREM: sum-distributes-over-plus
```
for (i,
     l,
     cond,
     'sum,
     list ('plus, g, h),
     a)
=   (for (i,
          l,
          cond,
          'sum,
          g,
          a)
       +  for (i,
               l,
               cond,
               'sum,
               h,
               a))
```

THEOREM: eval$-distributes-over-plus
```
eval$ (t, list ('plus, x, y), a) = (eval$ (t, x, a) + eval$ (t, y, a))
```

`; Section 3.4.  Window Dressings`

THEOREM: v&c$-list-defn
```
v&c$ ('list, l, va)
=   if l ≃ nil then nil
    else cons (v&c$ (t, car (l), va), v&c$ ('list, cdr (l), va)) endif
```

THEOREM: v&c$-defn
```
v&c$ (t, x, va)
=   if litatom (x) then cons (cdr (assoc (x, va)), 0)
```

```
    elseif $x \simeq$ nil then cons$(x, 0)$
    elseif car$(x)$ = 'quote then cons(cadr$(x)$, 0)
    else v&c-apply$(car$(x)$, v&c$('list, cdr$(x)$, $va$)) endif
```

```
; The following lemma has a hideous but trivial proof.  It generates
; 1222 cases!  The final stats are: [ 0.2 348.0 162.9 ] = 8.5 minutes.
```

THEOREM: eq-args-give-eq-values–apply-version

$$
\begin{aligned}
&((\mathit{fn} \neq \texttt{'quote})\\
&\ \wedge\ (\text{strip-cars}(\mathit{args1}) = \text{strip-cars}(\mathit{args2}))\\
&\ \wedge\ (\mathbf{f} \notin \mathit{args1})\\
&\ \wedge\ (\mathbf{f} \notin \mathit{args2}))\\
&\rightarrow\ ((\text{v\&c-apply\\$}(\mathit{fn}, \mathit{args1}) \leftrightarrow \text{v\&c-apply\\$}(\mathit{fn}, \mathit{args2}))\\
&\qquad \wedge\ (\text{car}(\text{v\&c-apply\\$}(\mathit{fn}, \mathit{args1})) = \text{car}(\text{v\&c-apply\\$}(\mathit{fn}, \mathit{args2}))))
\end{aligned}
$$

THEOREM: eq-args-give-eq-values

$$
\begin{aligned}
&((\mathit{fn} \neq \texttt{'quote})\\
&\ \wedge\ (\mathbf{f} \notin \text{v\&c\\$}(\texttt{'list}, \mathit{args1}, \mathit{va1}))\\
&\ \wedge\ (\mathbf{f} \notin \text{v\&c\\$}(\texttt{'list}, \mathit{args2}, \mathit{va2}))\\
&\ \wedge\ (\text{strip-cars}(\text{v\&c\\$}(\texttt{'list}, \mathit{args1}, \mathit{va1}))\\
&\qquad = \ \text{strip-cars}(\text{v\&c\\$}(\texttt{'list}, \mathit{args2}, \mathit{va2}))))\\
&\rightarrow\ ((\text{v\&c\\$}(\mathbf{t}, \text{cons}(\mathit{fn}, \mathit{args1}), \mathit{va1}) \leftrightarrow \text{v\&c\\$}(\mathbf{t}, \text{cons}(\mathit{fn}, \mathit{args2}), \mathit{va2}))\\
&\qquad \wedge\ (\text{car}(\text{v\&c\\$}(\mathbf{t}, \text{cons}(\mathit{fn}, \mathit{args1}), \mathit{va1}))\\
&\qquad\qquad = \ \text{car}(\text{v\&c\\$}(\mathbf{t}, \text{cons}(\mathit{fn}, \mathit{args2}), \mathit{va2}))))
\end{aligned}
$$

```
; Section 4.  Theorems about Partial Functions

; This section contains a series of events demonstrating a technique for proving
; theorems about interpreted functions.  The text was written 16 October, 1984,
; however it has been brought into conformance with the notation used in the
; quantifier paper.

; We would like to prove such theorems as
;     [(APPEND1 A B)] = (APPEND A B),
; where APPEND1 is a "partial" function whose body is exactly that of APPEND's with
; an APPEND1 in place of the APPEND call.

; The first question is which of the various equivalent formulations do we
; choose?

; It should be clear that the basic theorem we must prove is at the V&C level
; (V&C$ and V&C-APPLY$), not the EVAL level (EVAL$ and APPLY$).  EVAL recurses
; only through the s-expression it is given.  When given a concrete s-expression
; such as the one in the definition of APPEND1, the EVALs disappear and turn into
; function applications.  Function application at the EVAL level is performed by
; APPLY$ which defined in terms of V&C-APPLY$, which is in turn defined with V&C$
; of the BODY.  Thus, we see that the recursion needed to calculate the value of
; 'APPEND1 is actually carried out at the V&C level.  That is, it is V&C$, not EVAL$,
; that recurses on the body of 'APPEND1.   So we wish to formulate our basic lemma at
; the V&C$ level.

; To express the lemma we need we seem to have only two choices.  Either we
; express it in terms of the V&C$ of some typical 'APPEND1 expression, e.g.,
; (LIST 'APPEND1 E1 E2) or perhaps the body of 'APPEND1 itself; or we express it in
; terms of applying 'APPEND1 to some argument pairs with V&C-APPLY$.

; That is, we can use an expression like:

;      (V&C$ T (LIST 'APPEND1 E1 E2) A)

; or

;      (V&C$ T '(IF (LISTP X) (CONS & &) Y) *alist*) --

; or we can use an expression like:

;      (V&C-APPLY$ 'APPEND1 (LIST VC1 VC2)).

; I have found the V&C-APPLY$ route the simplest for the theorem prover.  The
; reason is that the V&C$ routes give rise to expressions which simplify by
; expanding the definition of V&C$ (or using our built in rewriters).  Let me
; describe the problem in more detail.

; My natural inclination was to use (V&C$ T '(IF (LISTP X) & &) a).  The idea of
; my proof was to have an induction hypothesis in which 'X is bound to (CDR X) in
; a and an induction conclusion in which 'X is bound to X in a.  Then I would
; expand the V&C$ in the conclusion, drive it through the body to the recursive
; call of 'APPEND1, open up that call to expose a V&C$ on the body of 'APPEND1 and observe
; that was my induction hypothesis.  But it was very messy because the natural
; inclination of the theorem prover to expand V&C$ on quoted s-expressions caused
; the inductive hypothesis to be rewritten to an IF and the inductive case was
; then splattered into several formulas.  The V&C$ in the conclusion naturally
; dove down to the recursive call as planned and was forcibly opened there to
; reveal the body.  But then the simplifier drove V&C$ through that before it
; could finally identify the induction hypothesis in it.  In short, by working at
; the V&C$ level on the body of the function we force the theorem prover to dive
; through the body 3 times:  once to put the hypothesis in normal form, once to
; drive the conclusion down to the recursive call we need to expand, and then
; after expanding the call to put the thing into normal form.

; We could turn off V&C$ to prevent it from expanding on the quoted function body
; in the inductive hypothesis.  However, we would then have to force manually its
; expansion on every subexpression of the body in the inductive conclusion.  The
; rather primitive proof checking commands at our disposal make it extremely
; awkward to expand V&C$ on a quoted constant in the conclusion without doing so
; in the hypothesis.  (The observation that the two calls of V&C$ in question
; have different alists permits one to distinguish them, but it as far as I know
; the only way to achieve the desired effect is to hint the expansion of V&C$ for
; every subexpression in the body.)

; The other V&C$ route, (V&C$ T (LIST 'APPEND1 E1 E2) A), suffers less extremely from
; the same tendency.  It expands into a V&C-APPLY$ both in the hypothesis and
; conclusion.  Manual intervention is required to open the V&C-APPLY$ in the
; conclusion to reveal a V&C$ of the body, but that V&C$ of the body is then
; automatically driven through the body and simplified to the same V&C-APPLY$
; expression to which the hypothesis simplified.  The only real problem with
; using (V&C$ T (LIST 'APPEND1 E1 E2) A) is that when we then turn to the question "Is
; (APPEND1 A B) equal to (APPEND A B)?" the lemma proved with V&C$ is useless as a
; rewrite rule.  That is because (APPEND1 A B) expands to a call of EVAL$ on the body
; and that is simplified to calls of APPLY$.  If APPLY$ is enabled, those calls
; become V&C-APPLY$ calls.  If V&C-APPLY$ were enabled, the V&C-APPLY$ calls
; would become calls of V&C$ on the body.  Only by driving through the body
; AGAIN would one arrive at a call of the form (V&C$ T (LIST 'APPEND1 E1 E2) A) to
; which the lemma would apply.

; I therefore decided to make my basic lemma be about V&C-APPLY$.  I leave V&C$
; enabled and V&C-APPLY$ disabled.  Because V&C-APPLY$ is disabled, my induction
; hypthesis stays put.  I have to force V&C-APPLY$ to expand ONCE in the
; induction conclusion to give rise to a V&C$ call on the body.  But then the
; enabled V&C$ is automatically driven through all quoted expressions giving rise
; to V&C-APPLY$ calls.  Those that are on subrs or total functions -- or
; functions about which we have already proved V&C-APPLY$ lemmas -- disappear,
; leaving only the V&C-APPLY$ calls of the "new" function being analyzed.  The
; induction hypothesis then hits those.  Once such a basic V&C-APPLY$ lemma is
; proved it is in the correct form to be used in subsequent proofs, both about
; V&C$ and about EVAL$.  In particular, the proof that (APPEND1 A B) is (APPEND A B)
; goes through immediately if APPLY$ is enabled and so converted to V&C-APPLY$.

; Most of the time V&C-APPLY$ is off but we would like V&C$ to walk through
; exprs composed of primitives.  So we "enable" V&C-APPLY$ on IF and
; SUBRs with the following two lemmas.  The variable and QUOTE cases are
; part of V&C$.
```

THEOREM: v&c-apply$-if

```
 v&c-apply$ ('if, args)
 =   if car (args)
     then if car (car (args))
          then fix-cost (car (cdr (args)), 1 + cdr (car (args)))
          else fix-cost (car (cdr (cdr (args))), 1 + cdr (car (args))) endif
     else '*1*false endif
```

THEOREM: v&c-apply$-subrp

```
 (subrp (fn) ∧ (fn ≠ 'if))
 →   (v&c-apply$ (fn, args)
       =   if '*1*false ∈ args  then '*1*false
           else cons (apply-subr (fn, strip-cars (args)),
                      1 + sum-cdrs (args)) endif)
```

```
; The basic observation used to prove non-termination is:
```

THEOREM: basic-nonterminating-lemma

(cdr (v&c-apply$ (*fn*, *args*)) $<$ *c*) $\rightarrow$ (v&c-apply$ (*fn*, *args*) $\neq$ cons (v, c))

```
; This lemma is really just the observation that X is not Y if their
; CDRs are different.  It's specialized to prevent its overuse.
; In addition, we convert "their CDRs are different" into
; "(CDR X) < (CDR Y)."

; It never occurs to the theorem prover to try to prove that x is non-F
; by proving (LISTP x).  We force it to make this consideration for
; V&C-APPLY$.
```

THEOREM: embarrassing-observation

listp (v&c-apply$ (*fn*, *args*)) $\rightarrow$ v&c-apply$ (*fn*, *args*)

```
; We are now prepared to prove the cited theorems about partial functions.

; We begin with APP.
```

DEFINITION:

```
app (x, y)
=   eval$ (t,
          '(if
          (equal x 'nil)
          y
          (cons (car x) (app (cdr x) y))),
        list (cons ('x, x), cons ('y, y)))
```

THEOREM: app-0-loops
v&c-apply$ ('app, list ('(0 . 1), cons (y, 0))) = **f**

```
; We are interested in the question, "What is (V&C-APPLY$ 'APP (LIST
; VC1 VC2))?"  (An earlier formulation of the lemma had explicit
; CONSes for the VCi above.  We discuss why that was inappropriate in
; the discussion of REVERSE1 below.)  VC1 and VC2 are typically pairs,
; containing the two lists to be appended in the CARs and the costs of
; computing them in their CDRs.  We must prove simultaneously that the
; above application returns a pair iff the first list is PROPERP and
; that the CAR of that pair is the desired answer.  [It is not
; strictly true that we must prove termination and correctness
; simultaneously in all cases.  See the footnote on COP below.]

; We will prove it by induction on the CAR of VC1.  When we expand the
; application of 'APP in the induction conclusion we will get a V&C$ of
; the body of 'APP.  The V&C$ will walk through the primitive function parts
; of the expression and will convert the recursive call of 'APP into
; another (V&C-APPLY$ 'APP (LIST vc1 vc2)).  We wish to have an inductive
; hypothesis about that application.  Thus, vc1 should be a pair whose
; CAR is (CDR (CAR VC1)) -- the CDR of the first arg -- and
; vc2 should be a pair whose CAR is the second arg.  What about the costs?
; From inspection of the body of 'APP we see the cost of computing the first
; arg to the recursion in 'APP is 1 and that of the second is 0.  Hence,
; we get the following rather odd induction hint:
```

DEFINITION:
app-induction-hint (*vc1*, *vc2*)
= **if** listp (car (*vc1*))
  then app-induction-hint (cons (cdr (car (*vc1*)), 1), cons (car (*vc2*), 0))
  else t endif

DEFINITION:
properp (x)
= **if** listp (x) **then** properp (cdr (x))
  else x = **nil endif**

THEOREM: app-is-partially-append–apply-version
(v&c-apply$ ('app, list (*vc1*, *vc2*)) $\leftrightarrow$ (*vc1* $\land$ *vc2* $\land$ properp (car (*vc1*))))

$\wedge$ (v&c-apply$ (`'app`, list ($vc1$, $vc2$))
    $\rightarrow$ (car (v&c-apply$ (`'app`, list ($vc1$, $vc2$)))
        $=$ append (car ($vc1$), car ($vc2$))))

THEOREM: app-is-partially-append
(v&c$ (**t**, `'(app x y)`, list (cons (`'x`, x), cons (`'y`, y))) $\leftrightarrow$ properp (x))
$\wedge$ (v&c$ (**t**, `'(app x y)`, list (cons (`'x`, x), cons (`'y`, y)))
    $\rightarrow$ (car (v&c$ (**t**, `'(app x y)`, list (cons (`'x`, x), cons (`'y`, y))))
        $=$ append (x, y)))

```
; The paper now turns to proving things about [(APPEND X Y)].  That is easy,
; because the functions in question are TAMEP:
```

THEOREM: append-x-y-is-tamep
v&c$ (**t**, `'(append x y)`, list (cons (`'x`, x), cons (`'y`, y)))
$\wedge$ (car (v&c$ (**t**, `'(append x y)`, list (cons (`'x`, x), cons (`'y`, y))))
    $=$ append (x, y))

```
; But the spirit of the remarks in the paper is to avoid the use of new
; (derived) rules of inference and construct the proofs from first
; principles.  To that end I define an interpreted version of APPEND,
; called APPEND1:
```

DEFINITION:
append1 (x, y)
$=$ eval$ (**t**,
```
          '(if
             (listp x)
             (cons (car x) (append1 (cdr x) y))
             y),
```
          list (cons (`'x`, x), cons (`'y`, y)))

```
; and using the same methodology as used in APP, construct the proof
; that it is TAMEP and is APPEND:
```

DEFINITION:
append1-induction-hint ($vc1$, $vc2$)
$=$ **if** listp (car ($vc1$))
    then append1-induction-hint (cons (cdr (car ($vc1$)), `1`), cons (car ($vc2$), `0`))
    else t endif

THEOREM: append1-is-tamep-and-is-append–apply-version

(v&c-apply$ (`'append1`, list (*vc1*, *vc2*)) $\leftrightarrow$ (*vc1* $\wedge$ *vc2*))
$\wedge$ (car (v&c-apply$ (`'append1`, list (*vc1*, *vc2*)))
= **if** *vc1* $\wedge$ *vc2* **then** append (car (*vc1*), car (*vc2*))
else `0` **endif**)

THEOREM: append1-x-y-is-tamep
v&c$ (**t**, `'(append x y)`, list (cons (`'x`, *x*), cons (`'y`, *y*)))
$\wedge$ (car (v&c$ (**t**, `'(append x y)`, list (cons (`'x`, *x*), cons (`'y`, *y*))))
= append (*x*, *y*))

```
; For the record, though I do not think it is mentioned in the paper,
; we can also prove:
```

THEOREM: append1-is-append
append1 (*x*, *y*) = append (*x*, *y*)

```
; Following the development in the paper, we now turn to:
```

DEFINITION: RUSSELL = eval$ (**t**, `'(not (russell))`, **nil**)

```
; The proof that 'RUSSELL does not terminate is now straightforward:
; just expand the definition of V&C-APPLY$ once to produce a V&C$,
; which walks through the formula and produces an identical call of
; V&C-APPLY$.
```

THEOREM: russell-does-not-terminate
v&c-apply$ (`'russell`, **nil**) = **f**

```
; Note that I use USE and not EXPAND.  EXPAND would replace the V&C-APPLY$
; by its definition.  USE adds the instanitated definition as a hypothesis.
; But under the assumption that (V&C-APPLY$ 'RUSSELL NIL) is not F, that
; hypothesis is contradictory.

; Consequently:
```

THEOREM: russell-is-f
RUSSELL = **f**

```
; The abstract of the paper claims we can prove facts about
; LEN, even though those facts aren't mentioned in the body
; of the paper.  This is a good time in the development of
; our methodology to do LEN:
```

DEFINITION:
len (x)
$=$ eval\$ (**t**,
`'(if (equal x 'nil) '0 (add1 (len (cdr x))))`,
list (cons (`'x`, x)))

```
; The definition of 'LEN terminates only on proper lists.  We
; can apply much the same program to such functions.

; We wish to prove that 'LEN is LENGTH when it terminates, and we
; want to characterize its termination.
```

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

```
; The key observation is that when applied to a 0 the function does
; not terminate.  In the application in question, namely, when we
; are proving the general fact about 'LEN and have to worry about
; recursion into a non-LISTP, the 0 arises by evaluating '(CDR X)
; and so has a cost of 1.  So, following the program carried out
; for RUSSELL, we observe that 'LEN doesn't terminate in that case:
```

THEOREM: len-loops-on-0
v&c-apply\$ (`'len`, list (cons (0, 1))) $=$ **f**

```
; We then follow the program carried out for 'APPEND1 and the others except
; our induction hint is sound (terminates on NLISTP not a NIL check).
```

DEFINITION:
len-induction-hint (vc)
$=$ **if** listp (car (vc)) **then** len-induction-hint (cons (cdr (car (vc)), 1))
else t endif

THEOREM: len-is-partially-length–apply-version
(v&c-apply\$ (`'len`, list (vc)) $\leftrightarrow$ ($vc \wedge$ properp (car (vc))))
$\wedge$ (v&c-apply\$ (`'len`, list (vc))
$\rightarrow$ (car (v&c-apply\$ (`'len`, list (vc))) $=$ length (car (vc))))

THEOREM: len-x-is-partially-length
(v&c\$ (**t**, `'(len x)`, list (cons (`'x`, x))) $\leftrightarrow$ properp (x))
$\wedge$ (v&c\$ (**t**, `'(len x)`, list (cons (`'x`, x)))
$\rightarrow$ (car (v&c\$ (**t**, `'(len x)`, list (cons (`'x`, x)))) $=$ length (x)))

```
; Note that all these proofs are exactly the same from the user's
; point of view:
; (0)  If the function is partial distill the reason it loops,
;      e.g., LEN-LOOPS-ON-0.
; (1)  Define the induction hint, which is mechanically derivable
;      from the interpreted definition, e.g., LEN-INDUCTION-HINT.
; (2)  Prove the termination and value properties about an arbitrary
;      V&C-APPLY$ of the function, e.g.,
;      LEN-IS-PARTIALLY-LENGTH--APPLY-VERSION.
; (3)  Prove the V&C$ version of the termination and value properties,
;      e.g., LEN-X-IS-PARTIALLY-LENGTH.

; We have done this for APPEND1 and LEN and will do it repeatedly
; for the other functions mentioned.  Ocassionally we have to prove
; lemmas to make the proofs go through, but the lemmas are not facts
; about V&C$ but just ordinary facts about the functions used in
; the definitions.  For example, when we prove F91 function we'll need
; some arithmetic, and when we deal with the various reverse functions
; we'll need some list processing stuff.

; Perhaps the key weakness in the programs developed so far is the
; concern about the cost of evaluating arguments as encoded in the
; induction hints.  The good news is that we are only concerned about
; the cost of evaluating the arguments of recursive calls of the
; function being considered, not the cost of all arguments.  (Consider
; that when we proved 'REVERSE1 correct we did not care how long
; 'REVERSE1 takes to produce the argument for 'AP, we merely cared how
; long 'CDR takes to produce the argument for 'REVERSE1.  That is, our
; concern with costs is focussed entirely in inductive hypotheses and
; not in the statement of our lemmas.)

; The bad news is that if the function being introduced recurses on some
; nonprimitive we seem to need lemmas telling us what the cost of
; the relevant computations are, if we are to follow this program.  The
; worse case seems to be when a function recurses on itself, as does
; TAK, for then we'd have to be deriving its cost as we go.  However, we
; do not have to compute the cost of an actual in closed form; we can use
; instead (CDR (V&C$ actual alist)) or its V&C-APPLY$ equivalent.

; To illustrate this, we attack the simple analogue of TAK, namely FID:
```

DEFINITION:
fid (x)

```
=   eval$ (t,
           '(if (zerop x) '0 (add1 (fid (fid (sub1 x))))),
           list (cons ('x, x)))
```

DEFINITION:
fid-induction-hint (*vc*)

```
=   if car (vc) ≃ 0 then t
    else fid-induction-hint (cons (car (vc) − 1, 1))
         ∧   fid-induction-hint (cons (car (vc) − 1,
                                       cdr (v&c-apply$ ('fid,
                                                        list (cons (car (vc) − 1,
                                                                    1)))))) endif
```

```
; Note that we provide two induction hypotheses, one for each call of FID.
; The first is simple:  the actual is one less than it was and the cost is 1.
; The second is complicated:  the actual is one less than it was and the
; cost is ... whatever it is!  The cost expression for the second hypothesis
; -- indeed for all the hypotheses generated -- can be obtained mechanically
; by just using the CDR of the V&C$ of the actual expression.  However, because
; FID recurses on itself, the induction hint for it cannot be written until
; one understands what the value delivered is.  If we tried to do with the
; value component what we did with the cost component, namely put in the
; CAR of the V&C$ of the actual expression, FID-INDUCTION-HINT would not be
; admissible.

; The only trouble with the above hint is that the second induction hypothesis
; will not be used because the outer application of 'FID is not on an
; explict CONS but on a (V&C-APPLY$ 'FID ...) that is known to be a list
; whose CAR is (SUB1 (CAR VC)).  We sort of wish we could do an elim and
; replace a LISTP by the CONS of its CAR and CDR, even for non-variables.
; That loops.  So we must be more devious.  The following lemma does an
; ELIM of a V&C-APPLY$ expression provided it occurs in a CONS.  That
; is the only case we are interested in since the V&C-APPLY$ in question
; is in the arglist of another V&C-APPLY$.
```

THEOREM: fake-elim

```
 v&c-apply$ (fn, args)
 →   (cons (v&c-apply$ (fn, args), tl)
         =   cons (cons (car (v&c-apply$ (fn, args)), cdr (v&c-apply$ (fn, args))),
                   tl))
```

```
; This lemma is dangerous because (CONS (CAR X) (CDR X)) = X can cause
; it to loop.  It doesn't if (CAR (V&C-APPLY$ FN ARGS)) is rewritten to
; something else first though!  If you try to prove termination without
; simultaneously proving a rewrite rule for the value, this lemma will
; bite you!

; We now prove the basic lemma for 'FID.
```

THEOREM: fid-is-tamep-and-id–apply-version

```
 (v&c-apply$ ('fid, list (vc)) ↔ vc)
 ∧    (car (v&c-apply$ ('fid, list (vc)))
       =   if vc  then fix (car (vc))
           else 0 endif)
```

THEOREM: fid-x-is-tamep

```
 v&c$ (t, '(fid x), list (cons ('x, x)))
 ∧    (car (v&c$ (t, '(fid x), list (cons ('x, x)))) = fix (x))
```

```
; Note that the program followed for 'FID is exactly that followed for
; 'APPEND1, except we have now generalized the costs specified in the induction
; hints, from closed forms such as 1 or 0, to the very expressions generated
; by V&C$.

; So now let's do something more complex, like GOPHER:
```

DEFINITION:

```
ro (x)
=    eval$ (t,
            '(cons
              (car (car x))
              (cons (cdr (car x)) (cdr x))),
            list (cons ('x, x)))
```

DEFINITION: rot (x) = cons (caar (x), cons (cdar (x), cdr (x)))

THEOREM: ro-is-tamep-and-rot–apply-version

```
 (v&c-apply$ ('ro, list (vc)) ↔ vc)
 ∧    (car (v&c-apply$ ('ro, list (vc)))
       =   if vc  then rot (car (vc))
           else 0 endif)
```

THEOREM: ro-x-is-tamep

```
 v&c$ (t, '(ro x), list (cons ('x, x)))
 ∧    (car (v&c$ (t, '(ro x), list (cons ('x, x)))) = rot (x))
```

DEFINITION:

gop (x)
= eval\$ (**t**,

```
'(if
  (or (nlistp x) (nlistp (car x)))
  x
  (gop (ro x))),
```

list (cons ('x, x)))

DEFINITION:
gopher (x)
= **if** $(x \simeq$ **nil**$) \vee$ (car $(x) \simeq$ **nil**) **then** x
else gopher (rot (x)) **endif**

DEFINITION:
gop-induction-hint (vc)
= **if** listp (car (vc)) $\wedge$ listp (caar (vc))
then gop-induction-hint (cons (rot (car (vc)),
cdr (v&c\$ (**t**,
'(ro x),
list (cons ('x, car (vc)))))))
else t endif

```
; Observe that the cost component above is just the CDR of the V&C$ of the
; actual expression.
```

THEOREM: gop-is-tamep-and-gopher–apply-version
(v&c-apply\$ ('gop, list (vc)) $\leftrightarrow$ vc)
$\wedge$ (car (v&c-apply\$ ('gop, list (vc)))
= **if** vc **then** gopher (car (vc))
else 0 **endif**)

THEOREM: gop-x-is-tamep
v&c\$ (**t**, '(gop x), list (cons ('x, x)))
$\wedge$ (car (v&c\$ (**t**, '(gop x), list (cons ('x, x)))) = gopher (x))

```
; It should be obvious that we do not need to think about the costs of
; interpreting these functions.

; We'll now do the 91 function.
```

DEFINITION:
f91 (x)
= eval\$ (**t**,

```
          '(if
             (lessp '100 x)
             (difference x '10)
             (f91 (f91 (plus x '11)))),
          list (cons ('x, x)))
```

DEFINITION:
g91 (x)
$=$ **if** $100 < x$ **then** $x - 10$
else 91 **endif**

THEOREM: equal-difference-0
$(x \not< y) \rightarrow ((y - x) = 0)$

DEFINITION:
f91-induction-hint (vc)

```
=   if 100 < car (vc) then t
    else f91-induction-hint (cons (car (vc) + 11,
                                   cdr (v&c$ (t,
                                              '(plus x '11),
                                              list (cons ('x, car (vc)))))))
         ∧  f91-induction-hint (cons (g91 (car (vc) + 11),
                                      cdr (v&c$ (t,
                                                 '(f91
                                                    (plus x
                                                     '11)),
                                                 list (cons ('x,
                                                             car (vc))))))) endif
```

```
; As for FID, the induction hint for F91 requires the knowledge of what
; the function returns.  Note however our cavalier treatment of the cost
; again.
```

THEOREM: f91-x-is-tamep-and-g91–apply-version
(v&c-apply$ ('f91, list (vc)) $\leftrightarrow$ vc)
$\wedge$ (car (v&c-apply$ ('f91, list (vc)))
$=$ **if** vc **then** g91 (car (vc))
else 0 **endif**)

THEOREM: f91-x-is-tamep
v&c$ (t, '(f91 x), list (cons ('x, x)))
$\wedge$ (car (v&c$ (t, '(f91 x), list (cons ('x, x)))) $=$ g91 (x))

```
; Footnote:  It is not necessary always to prove termination and
; correctness simultaneously.  The following trivial exercise demonstrates
; that they can be done separately sometimes.  However, in the case of
; a function like FID or just when a partial function is used in the
; definition of another function (e.g., if some function 'FOO called 'LEN
; on (FOO (CDR X))) it is necessary to know properties of the value delivered
; while one is proving termination.
```

DEFINITION:
cop (x)
= eval$ (**t**,
        `'(if (listp x) (cons (car x) (cop (cdr x))) x)`,
        list (cons (`'x`, x)))

DEFINITION:
cop-induction-hint (vc)
= **if** listp (car (vc)) **then** cop-induction-hint (cons (cdr (car (vc)), `1`))
  else t endif

THEOREM: v&c-apply$-cop-terminates
v&c-apply$ (`'cop`, list (vc)) $\leftrightarrow$ vc

THEOREM: v&c-apply$-cop-copies
car (v&c-apply$ (`'cop`, list (vc))) = car (vc)

```
;  We complete the exercises on interpreting inadmissible functions
;  by showing that Burstall's one argument reverse function is total
;  and computes REVERSE.
```

DEFINITION:
rv (l)
= eval$ (**t**,

```
'(if
  (nlistp l)
  'nil
  (if
   (nlistp (cdr l))
   (cons (car l) 'nil)
   (cons
    (car (rv (cdr l)))
    (rv (cons (car l) (rv (cdr (rv (cdr l))))))))),
```

list (cons (`'l`, l)))

DEFINITION:

```
reverse (x)
=    if listp (x)  then append (reverse (cdr (x)), list (car (x)))
     else nil endif
```

EVENT: Disable fake-elim.

THEOREM: length-reverse
length (reverse (l)) = length (l)

THEOREM: listp-reverse
listp (reverse (l)) = listp (l)

THEOREM: length-cdr
listp (l) → (length (cdr (l)) < length (l))

DEFINITION:

```
rv-induction-hint (vc)
=    if car (vc) ≃ nil  then t
     elseif cdr (car (vc)) ≃ nil  then t
     else rv-induction-hint (cons (cdr (car (vc)), 1))
          ∧   rv-induction-hint (cons (cdr (reverse (cdr (car (vc)))),
                                       cdr (v&c$ (t,
                                                  '(cdr
                                                    (rv (cdr l))),
                                                  list (cons ('l,
                                                              car (vc)))))))
          ∧   rv-induction-hint (cons (cons (car (car (vc)),
                                             reverse (cdr (reverse (cdr (car (vc)))))),
                                       cdr (v&c$ (t,
                                                  '(cons
                                                    (car l)
                                                    (rv
                                                     (cdr
                                                      (rv
                                                       (cdr l))))),
                                                  list (cons ('l,
                                                              car (vc))))))) endif
```

THEOREM: properp-reverse
properp (reverse (x))

THEOREM: reverse-reverse
properp (x) → (reverse (reverse (x)) = x)

THEOREM: properp-cdr
(listp (x) ∧ properp (x)) → properp (cdr (x))

THEOREM: fake-elim2
$((\text{car}\,(\text{v\&c-apply\$}\,(\mathit{fn},\ \mathit{args})) = v) \wedge \text{listp}\,(\text{v\&c-apply\$}\,(\mathit{fn},\ \mathit{args})))$
$\rightarrow\ \ (\text{cons}\,(\text{v\&c-apply\$}\,(\mathit{fn},\ \mathit{args}),\ \mathit{tl})$
$\quad = \ \ \text{cons}\,(\text{cons}\,(v,\ \text{cdr}\,(\text{v\&c-apply\$}\,(\mathit{fn},\ \mathit{args}))),\ \mathit{tl}))$

THEOREM: plus-1
$(\mathtt{1} + x) = (1 + x)$

THEOREM: numberp-cdr-v&c$
$\text{cdr}\,(\text{v\&c\$}\,(\mathbf{t},\ x,\ a)) \in \mathbf{N}$

THEOREM: numberp-cdr-v&c-apply$
$\text{cdr}\,(\text{v\&c-apply\$}\,(\mathit{fn},\ \mathit{args})) \in \mathbf{N}$

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

```
;  When trying to prove anything about the body of 'RV the theorem
;  prover spends an inordinant amount of time.  To speed things up
;  we prove the following sequence of 4 lemmas, the last of which
;  permits us to simplify (V&C$ T body alist) to its normal form
;  in one step.  The preceding 3 lemmas speed the proof of the last.
;  I think technically these 4 lemmas are unnecessary.  Except for
;  these "short-cut" lemmas the program followed for RV is exactly
;  the same as that followed for LEN and APPEND1.
```

THEOREM: short-cut1
```
 v&c$ (t, '(rv (cdr (rv (cdr l)))), list (cons ('l, l)))
 =  if v&c-apply$ ('rv, list (cons (cdr (l), 1)))
    then v&c-apply$ ('rv,
                    list (cons (cdar (v&c-apply$ ('rv, list (cons (cdr (l), 1)))),
                                1 + cdr (v&c-apply$ ('rv,
                                                     list (cons (cdr (l), 1)))))))
    else f endif
```

THEOREM: short-cut2
```
 v&c$ (t,
       '(rv (cons (car l) (rv (cdr (rv (cdr l)))))),
       list (cons ('l, l)))
 =  if v&c-apply$ ('rv, list (cons (cdr (l), 1)))
    then if v&c-apply$ ('rv,
                        list (cons (cdar (v&c-apply$ ('rv,
                                                      list (cons (cdr (l), 1)))),
                                    1 + cdr (v&c-apply$ ('rv,
                                                  list (cons (cdr (l), 1)))))))
     then v&c-apply$ ('rv,
                    list (cons (cons (car (l),
                                      car (v&c-apply$ ('rv,
                                                       list (cons (cdar (v&c-apply$ ('rv,
                                                                                     list (cons (cdr (l),
                                                                                                1)))),
                                                                   1 + cdr (v&c-apply$ ('rv,
                                                                                        list (cons (cdr (l
                                                                                                   1)))))
                              1 + (1 + cdr (v&c-apply$ ('rv,
                                                       list (cons (cdar (v&c-apply$ ('rv,
                                                                                     list (cons (cdr (
                                                                                                1))))
                                                                   1 + cdr (v&c-apply$ ('rv,
                                                                                        list (cons (cd
                                                                                                   1)

     else f endif
  else f endif
```

```
(PROVE-LEMMA SHORT-CUT3
  (REWRITE)
  (EQUAL
   (V&C$ T
         '(CONS (CAR (RV (CDR L)))
                (RV (CONS (CAR L) (RV (CDR (RV (CDR L)))))))
         (LIST (CONS 'L L)))
   (IF (V&C-APPLY$ 'RV
                   (LIST (CONS (CDR L) 1)))
    (IF (V&C-APPLY$ 'RV
                 (LIST (CONS (CDAR (V&C-APPLY$ 'RV
                                                (LIST (CONS (CDR L) 1))))
                             (ADD1 (CDR (V&C-APPLY$ 'RV
                                                     (LIST (CONS (CDR L) 1))))))))
     (IF      (V&C-APPLY$ 'RV
      (LIST
       (CONS
        (CONS
         (CAR L)
         (CAR
          (V&C-APPLY$ 'RV
             (LIST (CONS (CDAR (V&C-APPLY$ 'RV
                                           (LIST (CONS (CDR L) 1))))
                         (ADD1 (CDR (V&C-APPLY$ 'RV
                                                (LIST (CONS (CDR L) 1))))))))))
        (ADD1
         (ADD1
          (CDR
           (V&C-APPLY$ 'RV
             (LIST (CONS (CDAR (V&C-APPLY$ 'RV
                                           (LIST (CONS (CDR L) 1))))
                         (ADD1 (PLUS (CDR (V&C-APPLY$ 'RV
                                                      (LIST (CONS (CDR L) 1))))
                                     0)))))))))))
    (CONS
     (CONS
      (CAAR (V&C-APPLY$ 'RV
                        (LIST (CONS (CDR L) 1))))
      (CAR
       (V&C-APPLY$ 'RV
        (LIST
         (CONS
          (CONS
           (CAR L)
           (CAR
            (V&C-APPLY$ 'RV
             (LIST (CONS (CDAR (V&C-APPLY$ 'RV
                                           (LIST (CONS (CDR L) 1))))
                         (ADD1 (CDR (V&C-APPLY$ 'RV
                                                (LIST (CONS (CDR L) 1))))))))))
          (ADD1
           (ADD1
            (CDR
             (V&C-APPLY$ 'RV
              (LIST
                 (CONS
                     (CDAR (V&C-APPLY$ 'RV
                                     (LIST (CONS (CDR L) 1))))
                     (ADD1 (CDR (V&C-APPLY$ 'RV
                                            (LIST (CONS (CDR L) 1))))))))))))))))
     (ADD1
      (ADD1
       (PLUS
        (CDR (V&C-APPLY$ 'RV
                         (LIST (CONS (CDR L) 1))))
        (CDR
         (V&C-APPLY$ 'RV
          (LIST
           (CONS
            (CONS
             (CAR L)
             (CAR
              (V&C-APPLY$ 'RV
               (LIST
                   (CONS (CDAR (V&C-APPLY$ 'RV
                                         (LIST (CONS (CDR L) 1))))
                         (ADD1 (CDR (V&C-APPLY$ 'RV
                                              (LIST (CONS (CDR L) 1))))))))))
            (ADD1
             (ADD1
              (CDR
               (V&C-APPLY$ 'RV
                (LIST
                 (CONS
                  (CDAR (V&C-APPLY$ 'RV
                                   (LIST (CONS (CDR L) 1))))
                  (ADD1
                      (CDR (V&C-APPLY$ 'RV
                                      (LIST (CONS (CDR L) 1)))))))))))))))
    F)
     F)
    F)))



(PROVE-LEMMA SHORT-CUT4
  (REWRITE)
  (EQUAL
   (V&C$ T
         '(IF (NLISTP L)
              'NIL
              (IF (NLISTP (CDR L))
                  (CONS (CAR L) 'NIL)
                  (CONS (CAR (RV (CDR L)))
                        (RV (CONS (CAR L)
                                  (RV (CDR (RV (CDR L)))))))))
         (LIST (CONS 'L L)))
   (IF
    (NLISTP L)
    (CONS NIL 2)
```

```
(IF
 (NLISTP (CDR L))
 (CONS (LIST (CAR L)) 7)
   (IF (V&C-APPLY$ 'RV
              (LIST (CONS (CDR L) 1)))
(IF (V&C-APPLY$ 'RV
           (LIST (CONS (CDAR (V&C-APPLY$ 'RV
                                        (LIST (CONS (CDR L) 1))))
                        (ADD1 (CDR (V&C-APPLY$ 'RV
                                             (LIST (CONS (CDR L) 1))))))))
 (IF      (V&C-APPLY$ 'RV
  (LIST
   (CONS
    (CONS
     (CAR L)
     (CAR
      (V&C-APPLY$ 'RV
         (LIST (CONS (CDAR (V&C-APPLY$ 'RV
                                     (LIST (CONS (CDR L) 1))))
                     (ADD1 (CDR (V&C-APPLY$ 'RV
                                          (LIST (CONS (CDR L) 1))))))))))
    (ADD1
     (ADD1
      (CDR
       (V&C-APPLY$ 'RV
         (LIST (CONS (CDAR (V&C-APPLY$ 'RV
                                     (LIST (CONS (CDR L) 1))))
                     (ADD1 (PLUS (CDR (V&C-APPLY$ 'RV
                                                (LIST (CONS (CDR L) 1))))
                                 0)))))))))))
(CONS
 (CONS
  (CAAR (V&C-APPLY$ 'RV
                  (LIST (CONS (CDR L) 1))))
  (CAR
   (V&C-APPLY$ 'RV
    (LIST
     (CONS
      (CONS
       (CAR L)
       (CAR
        (V&C-APPLY$ 'RV
         (LIST (CONS (CDAR (V&C-APPLY$ 'RV
                                     (LIST (CONS (CDR L) 1))))
                     (ADD1 (CDR (V&C-APPLY$ 'RV
                                            (LIST (CONS (CDR L) 1))))))))))
      (ADD1
       (ADD1
        (CDR
         (V&C-APPLY$ 'RV
          (LIST
           (CONS
               (CDAR (V&C-APPLY$ 'RV
                                 (LIST (CONS (CDR L) 1))))
               (ADD1 (CDR (V&C-APPLY$ 'RV
                                      (LIST (CONS (CDR L) 1))))))))))))))))
 (ADD1 (ADD1 (ADD1 (ADD1 (ADD1
 (ADD1
  (ADD1
   (PLUS
    (CDR (V&C-APPLY$ 'RV
                     (LIST (CONS (CDR L) 1))))
    (CDR
     (V&C-APPLY$ 'RV
      (LIST
       (CONS
        (CONS
         (CAR L)
         (CAR
          (V&C-APPLY$ 'RV
           (LIST
               (CONS (CDAR (V&C-APPLY$ 'RV
                                       (LIST (CONS (CDR L) 1))))
                     (ADD1 (CDR (V&C-APPLY$ 'RV
                                            (LIST (CONS (CDR L) 1))))))))))
        (ADD1
         (ADD1
          (CDR
           (V&C-APPLY$ 'RV
            (LIST
             (CONS
              (CDAR (V&C-APPLY$ 'RV
                                (LIST (CONS (CDR L) 1))))
              (ADD1
                  (CDR (V&C-APPLY$ 'RV
                                   (LIST (CONS (CDR L) 1))))))))))))))))))))))))))
F)
 F)
```

```
    F)
     ))))
```

THEOREM: rv-is-tamep-and-reverse–apply-version
(v&c-apply\$ (`'rv`, list (vc)) $\leftrightarrow$ vc)
$\wedge$ (car (v&c-apply\$ (`'rv`, list (vc)))
    $=$ **if** vc **then** reverse (car (vc))
        else 0 **endif**)

THEOREM: rv-x-is-tamep
v&c\$ (**t**, `'(rv x)`, list (cons (`'x`, x)))
$\wedge$ (car (v&c\$ (**t**, `'(rv x)`, list (cons (`'x`, x)))) $=$ reverse (x))

```
; Section 5.  Proofs about Partial Functions.

; Many of the theorems noted in this section were proved above.
; However, we here prove the "metatheorems" cited.

; The following formula was not a theorem until we added
; the new axiom of (SUBRP 'QUOTE) = F.  The need for this was
; not known until we tried to prove the formula for the paper.
```

THEOREM: definedness-condition-for-subrp
(($fn \neq$ `'if`) $\wedge$ subrp (fn))
$\rightarrow$ (v&c\$ (**t**, cons (fn, $args$), a) $\leftrightarrow$ (**f** $\notin$ v&c\$ (`'list`, $args$, a)))

THEOREM: value-theorem-for-subrp-lemma
(caddr (strip-cars (l)) $=$ caaddr (l))
$\wedge$ (cadr (strip-cars (l)) $=$ caadr (l))
$\wedge$ (car (strip-cars (l)) $=$ caar (l))

```
; The following formula wasn't a theorem until we added the
; axiom (SUBRP 'QUOTE) = F.
```

THEOREM: value-theorem-for-subrp
(subrp (fn) $\wedge$ v&c\$ (**t**, cons (fn, $args$), a))
$\rightarrow$ (car (v&c\$ (**t**, cons (fn, $args$), a))
    $=$ apply-subr (fn, strip-cars (v&c\$ (`'list`, $args$, a))))

```
; !!! BUG.  I HAD TO ADD THE STRIP-CARS TO THE RHS BELOW!
; !!! BUG.  I HAD TO THE ADD THE FN /= 'QUOTE BELOW!
```

THEOREM: cost-theorem-for-non-subrp

$$
\begin{array}{l}
((fn \neq \texttt{'quote}) \land (\neg\ \mathrm{subrp}\,(fn)) \land \mathrm{v\&c\$}\,(\mathbf{t},\ \mathrm{cons}\,(fn,\ args),\ a)) \\
\rightarrow\quad (\mathrm{cdr}\,(\mathrm{v\&c\$}\,(\mathbf{t},\ \mathrm{cons}\,(fn,\ args),\ a)) \\
\qquad >\quad \mathrm{cdr}\,(\mathrm{v\&c\$}\,(\mathbf{t}, \\
\qquad\qquad\qquad\qquad \mathrm{body}\,(fn), \\
\qquad\qquad\qquad\qquad \mathrm{pairlist}\,(\mathrm{formals}\,(fn), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{strip\text{-}cars}\,(\mathrm{v\&c\$}\,(\texttt{'list},\ args,\ a))))))
\end{array}
$$

THEOREM: append1-is-tame

$$
\begin{array}{l}
(\mathrm{v\&c\$}\,(\mathbf{t},\ u,\ a) \land \mathrm{v\&c\$}\,(\mathbf{t},\ v,\ a)) \\
\rightarrow\quad (\mathrm{v\&c\$}\,(\mathbf{t},\ \mathrm{list}\,(\texttt{'append1},\ u,\ v),\ a) \\
\qquad \land\quad (\mathrm{car}\,(\mathrm{v\&c\$}\,(\mathbf{t},\ \mathrm{list}\,(\texttt{'append1},\ u,\ v),\ a)) \\
\qquad\qquad =\quad \mathrm{append}\,(\mathrm{car}\,(\mathrm{v\&c\$}\,(\mathbf{t},\ u,\ a)),\ \mathrm{car}\,(\mathrm{v\&c\$}\,(\mathbf{t},\ v,\ a)))))
\end{array}
$$

```
; We now prove that REVERSE is TAMEP from the fact that APPEND is.
; Once again, we introduce REVERSE1 so as not to use the built-in
; TAMEP rules.

; It was in this proof that we learned that it is better to consider
;      (V&C-APPLY 'APPEND1 (LIST VC1 VC2))
; than
;      (V&C-APPLY 'APPEND1 (LIST (CONS A C1) (CONS B C2))).
; Our first use of V&C-APPLY$ in these basic lemmas contained CONSes
; as shown above.  This made the induction hint easier because we could
; instantiate A with (CDR A), C1 with 1, etc., instead of using that
; horrid CONS.  Unfortunately, no term of the form
;      (V&C-APPLY 'APPEND1 (LIST (CONS A C1) (CONS B C2))).
; arises when proving facts about 'REVERSE1 because the 'APPEND1 call in 'REVERSE1
; becomes
;      (V&C-APPLY 'APPEND1 (LIST (V&C-APPLY$ 'REVERSE1 &) &))
; where (V&C-APPLY$ 'REVERSE1 ...) is known to be LISTP.  Therefore, we
; phrase our basic lemmas in the form shown:
;      (V&C-APPLY 'APPEND1 (LIST VC1 VC2))
```

DEFINITION:

reverse1 (x)

$=$ eval\$ $(\mathbf{t},$

```
        '(if
          (listp x)
          (append1
           (reverse1 (cdr x))
           (cons (car x) 'nil))
          'nil),
```

list (cons (`'x`, x)))

Definition:
reverse1-induction-hint (vc)

```
=  if listp (car (vc))  then reverse1-induction-hint (cons (cdr (car (vc)), 1))
   else t endif
```

Theorem: reverse1-is-tamep-and-reverse–apply-version
(v&c-apply$ (`'reverse1`, list (vc)) $\leftrightarrow$ vc)
$\land$ (car (v&c-apply$ (`'reverse1`, list (vc)))
= **if** vc **then** reverse (car (vc))
else 0 **endif**)

Theorem: reverse1-x-is-tamep
v&c$ (**t**, `'(reverse1 x)`, list (cons (`'x`, x)))
$\land$ (car (v&c$ (**t**, `'(reverse1 x)`, list (cons (`'x`, x)))) = reverse (x))

```
; Section 6.  EVAL and APPLY

; The use of FREE-VARS in the paper suggests it is a fn of one argument,
; but it actually is mutually recursive with FREE-VARS-LIST and we handle
; it in the expected way.  I don't bother to define the one argument
; sugar coating and prove theorems more general than those given in the
; paper.
```

Definition:
free-vars (flg, x)

```
=  if flg = 'list
   then if x ≃ nil then nil
          else free-vars (t, car (x)) ∪ free-vars ('list, cdr (x)) endif
   elseif litatom (x)  then cons (x, nil)
   elseif x ≃ nil  then nil
   elseif car (x) = 'quote  then nil
   else free-vars ('list, cdr (x)) endif
```

Theorem: member-union
$(a \in (x \cup y)) = ((a \in x) \lor (a \in y))$

Theorem: irrelevant-bindings-can-be-deleted
($v \notin$ free-vars (flg, x))
$\rightarrow$ (eval$ (flg, x, cons (cons (v, val), a)) = eval$ (flg, x, a))

Definition:
substitute (x, y, flg, z)

```
=   if $flg$ = 'list
    then if $z \simeq$ nil then nil
         else cons (substitute ($x$, $y$, t, car ($z$)),
                    substitute ($x$, $y$, 'list, cdr ($z$))) endif
    elseif $y = z$ then $x$
    elseif litatom ($z$) then $z$
    elseif $z \simeq$ nil then $z$
    elseif car ($z$) = 'quote then $z$
    else cons (car ($z$), substitute ($x$, $y$, 'list, cdr ($z$))) endif
```

```
; I interchanged the names of X and Y in transcribing this theorem
; to bring them into accordance with the names used in the defn above.
```

THEOREM: substitution-of-equal-eval$s

```
 (eval$ (t, $x$, $a$) = eval$ (t, $y$, $a$))
 →   (eval$ ($flg$, substitute ($x$, $y$, $flg$, $z$), $a$) = eval$ ($flg$, $z$, $a$))
```

```
; Section 8.  Theorems about Quantifiers

; The following lemmas suggest an improvement to TYPE-SET.
; These lemmas are not mentioned in the paper but some are
; used in subsequent proofs.
```

THEOREM: for-always-is-boolean

```
 for ($x$,
      $y$,
      $cond$,
      'always,
      $body$,
      $alist$)
 →   (for ($x$,
           $y$,
           $cond$,
           'always,
           $body$,
           $alist$)
       =   t)
```

THEOREM: for-exists-is-boolean

```
 for ($x$,
      $y$,
      $cond$,
      'exists,
```

body,
alist)

→ (for (*x*,
y,
cond,
`'exists`,
body,
alist)
= **t**)

Theorem: numberp-for-sum
for (*x*,
y,
cond,
`'sum`,
body,
alist)$\in$ **N**

Theorem: numberp-for-multiply
for (*x*,
y,
cond,
`'multiply`,
body,
alist)$\in$ **N**

Theorem: numberp-for-count
for (*x*,
y,
cond,
`'count`,
body,
alist)$\in$ **N**

Theorem: numberp-for-max
for (*x*,
y,
cond,
`'max`,
body,
alist)$\in$ **N**

```
; Now I begin the classical quantifier theorems...
```

THEOREM: assoc-of-append
append (append (a, b), c) = append (a, append (b, c))

THEOREM: for-append-collect
```
 for (x,
      append (a, b),
      cond,
      'collect,
      body,
      alist)
 =    append (for (x,
                   a,
                   cond,
                   'collect,
                   body,
                   alist),
              for (x,
                   b,
                   cond,
                   'collect,
                   body,
                   alist))
```

THEOREM: for-append-count
```
 for (x,
      append (a, b),
      cond,
      'count,
      body,
      alist)
 =    (for (x,
            a,
            cond,
            'count,
            body,
            alist)
       +    for (x,
                 b,
                 cond,
                 'count,
                 body,
                 alist))
```

THEOREM: for-append-add-to-set
```
 for (x,
       append ($a$, $b$),
       $cond$,
       'add-to-set,
       $body$,
       $alist$)
 =     (for ($x$,
            $a$,
            $cond$,
            'add-to-set,
            $body$,
            $alist$)
        ∪   for ($x$,
                 $b$,
                 $cond$,
                 'add-to-set,
                 $body$,
                 $alist$))
```

THEOREM: for-append-do-return

```
 for ($x$,
       append ($a$, $b$),
       $cond$,
       'do-return,
       $body$,
       $alist$)
 =     **if** for ($x$,
              $a$,
              list ('quote, **t**),
              'exists,
              $cond$,
              $alist$)
       **then** for ($x$,
                 $a$,
                 $cond$,
                 'do-return,
                 $body$,
                 $alist$)
       **else** for ($x$,
                 $b$,
                 $cond$,
                 'do-return,
                 $body$,
                 $alist$) **endif**
```

```
;  I am going to prove this by proving each case separately and
;  then combining them...

;(PROVE-LEMMA FOR-APPEND-ALWAYS-MAX-EXISTS-SUM-APPEND-MULTIPLY-UNION NIL
;  (IMPLIES (MEMBER OP '(ALWAYS MAX EXISTS SUM
;                               APPEND MULTIPLY UNION))
;           (EQUAL
;             (FOR X (APPEND A B) COND OP BODY ALIST)
;             (QUANTIFIER-OPERATION OP
;                                   (FOR X A COND OP BODY ALIST)
;                                   (FOR X B COND OP BODY ALIST)))))
```

THEOREM: for-append-always

```
 for (x,
      append (a, b),
      cond,
      'always,
      body,
      alist)
 =    quantifier-operation ('always,
                            for (x,
                                 a,
                                 cond,
                                 'always,
                                 body,
                                 alist),
                            for (x,
                                 b,
                                 cond,
                                 'always,
                                 body,
                                 alist))
```

THEOREM: for-append-max

```
 for (x,
      append (a, b),
      cond,
      'max,
      body,
      alist)
 =    quantifier-operation ('max,
                            for (x,
                                 a,
                                 cond,
                                 'max,
                                 body,
                                 alist),
                            for (x,
                                 b,
                                 cond,
                                 'max,
                                 body,
                                 alist))
```

THEOREM: for-append-exists

```
for (x,
     append (a, b),
     cond,
     'exists,
     body,
     alist)
=    quantifier-operation ('exists,
                               for (x,
                                    a,
                                    cond,
                                    'exists,
                                    body,
                                    alist),
                               for (x,
                                    b,
                                    cond,
                                    'exists,
                                    body,
                                    alist))
```

THEOREM: for-append-sum

```
for (x,
     append (a, b),
     cond,
     'sum,
     body,
     alist)
=    quantifier-operation ('sum,
                               for (x,
                                    a,
                                    cond,
                                    'sum,
                                    body,
                                    alist),
                               for (x,
                                    b,
                                    cond,
                                    'sum,
                                    body,
                                    alist))
```

Theorem: for-append-append

```
for (x,
     append (a, b),
     cond,
     'append,
     body,
     alist)
=    quantifier-operation ('append,
                          for (x,
                                a,
                                cond,
                                'append,
                                body,
                                alist),
                          for (x,
                                b,
                                cond,
                                'append,
                                body,
                                alist))
```

Theorem: assoc-of-times

$((i * j) * k) = (i * (j * k))$

Theorem: for-append-multiply

```
for (x,
     append (a, b),
     cond,
     'multiply,
     body,
     alist)
=    quantifier-operation ('multiply,
                          for (x,
                                a,
                                cond,
                                'multiply,
                                body,
                                alist),
                          for (x,
                                b,
                                cond,
                                'multiply,
                                body,
                                alist))
```

THEOREM: assoc-of-union
$((a \cup b) \cup c) = (a \cup (b \cup c))$

THEOREM: for-append-union

```
for (x,
     append (a, b),
     cond,
     'union,
     body,
     alist)
=    quantifier-operation ('union,
                           for (x,
                                a,
                                cond,
                                'union,
                                body,
                                alist),
                           for (x,
                                b,
                                cond,
                                'union,
                                body,
                                alist))
```

THEOREM: for-append-always-max-exists-sum-append-multiply-union

```
(op ∈ '(always max exists sum append multiply union))
→    (for (x,
           append (a, b),
           cond,
           op,
           body,
           alist)
      =    quantifier-operation (op,
                                 for (x,
                                      a,
                                      cond,
                                      op,
                                      body,
                                      alist),
                                 for (x,
                                      b,
                                      cond,
                                      op,
                                      body,
                                      alist)))
```

Theorem: for-sum-plus

```
for (x,
     r,
     cond,
     'sum,
     list ('plus, g, h),
     alist)
=   (for (x,
          r,
          cond,
          'sum,
          g,
          alist)
      +  for (x,
              r,
              cond,
              'sum,
              h,
              alist))
```

Theorem: times-0

$(y \simeq 0) \rightarrow ((x * y) = 0)$

Theorem: times-add1

$(x * (1 + y)) = (x + (x * y))$

Theorem: commut-of-times

$(x * y) = (y * x)$

Theorem: commut2-of-times

$(x * (y * z)) = (y * (x * z))$

Theorem: for-multiply-times

```
for (x,
     r,
     cond,
      'multiply,
      list ('times, g, h),
      alist)
 =   (for (x,
           r,
           cond,
           'multiply,
           g,
           alist)
       *   for (x,
                r,
                cond,
                'multiply,
                h,
                alist))
```

Theorem: for-always-and

```
 for (x,
      r,
      cond,
      'always,
      list ('and, g, h),
      alist)
 =   (for (x,
           r,
           cond,
           'always,
           g,
           alist)
       ∧   for (x,
                r,
                cond,
                'always,
                h,
                alist))
```

Theorem: for-exists-or

```
 for (x,
      r,
      cond,
      'exists,
      list ('or, g, h),
      alist)
 =   (for (x,
            r,
            cond,
            'exists,
            g,
            alist)
      ∨   for (x,
                r,
                cond,
                'exists,
                h,
                alist))
```

THEOREM: unused-indicial-can-be-dropped-from-body

```
 (x ∉ free-vars (t, body))
 →   (for (x,
           r,
           cond,
           op,
           body,
           alist)
       =   for (x,
                r,
                cond,
                op,
                list ('quote, eval$ (t, body, alist)),
                alist))
```

THEOREM: unused-indicial-can-be-dropped-from-cond

```
 (x ∉ free-vars (t, cond))
 →   (for (x,
           r,
           cond,
           op,
           body,
           alist)
       =   for (x,
                r,
                list ('quote, eval$ (t, cond, alist)),
                op,
                body,
                alist))
```

THEOREM: for-sum-constant

```
 for (x,
      r,
     cond,
     'sum,
     list ('quote, body),
     alist)
=   (body * for (x,
                 r,
                 list ('quote, t),
                 'count,
                 cond,
                 alist))
```

```
; The following theorems about constant bodies are proved
; but not mentioned in the paper.
```

THEOREM: for-count-constant

```
for (x,
     r,
     list ('quote, t),
     'count,
     list ('quote, body),
     alist)
=   if body then length (r)
    else 0 endif
```

DEFINITION:

```
exp (i, j)
=  if j ≃ 0 then 1
   else i * exp (i, j - 1) endif
```

THEOREM: for-multiply-constant

```
for (x,
     r,
     cond,
     'multiply,
     list ('quote, body),
     alist)
=   exp (body,
         for (x,
              r,
              list ('quote, t),
              'count,
              cond,
              alist))
```

THEOREM: max-constant

```
for (x,
     r,
     cond,
     'max,
     list ('quote, body),
     alist)
=    if for (x,
             r,
             list ('quote, t),
             'exists,
             cond,
             alist) then fix (body)
     else 0 endif
```

```
; The next two lemmas are interesting hacks.  Consider EXISTS-CONSTANT
; below.  The first rewrite rule in it applies to
; quoted bodies and moves the arbitrary cond into the body of a new
; exists.  This would loop forever by itself if the cond were itself
; quoted.  However, to stop that loop we prove the second rewrite rule
; which addresses itself only to the case of a quoted body and a true
; cond clause.  The second rule could not be proved as a separate
; event after the first because the first makes it loop forever.
; If the second is proved before the first, the resulting set of
; rewrite rules loops forever because the first one -- proved last --
; has priority.  By proving them simultaneously and putting the second
; rule into the second conjunct we arrange for it to be the first
; of the two rules tried and for it to be proved before the other rule
; is available.  Crock, Crock, Crock.
```

THEOREM: always-constant

```
(for (x,
      r,
      cond,
      'always,
      list ('quote, body),
      alist)
 =    (body ∨ for (x,
                   r,
                   list ('quote, t),
                   'always,
                   list ('not, cond),
                   alist)))
```

```
∧   (for (x,
          r,
          list ('quote, t),
          'always,
          list ('quote, body),
          alist)
      =   (body ∨ (r ≃ nil)))
```

Theorem: exists-constant

```
(for (x,
      r,
      cond,
      'exists,
      list ('quote, body),
      alist)
 =   (body ∧ for (x,
                  r,
                  list ('quote, t),
                  'exists,
                  cond,
                  alist)))
∧   (for (x,
          r,
          list ('quote, t),
          'exists,
          list ('quote, body),
          alist)
      =   (body ∧ listp (r)))
```

Theorem: add-to-set-constant

```
for (x,
     r,
     cond,
     'add-to-set,
     list ('quote, body),
     alist)
=   if for (x,
            r,
            list ('quote, t),
            'exists,
            cond,
            alist) then list (body)
    else nil endif
```

Theorem: do-return-constant

```
for (x,
     r,
     cond,
     'do-return,
     list ('quote, body),
     alist)
=    if for (x,
             r,
             list ('quote, t),
             'exists,
             cond,
             alist) then body
     else nil endif
```

```
; The only quantifiers for which we do not have a theorem about
; a quoted body are APPEND COLLECT and UNION for which I could not
; think of a lemma.
```

```
; The next three are not mentioned in the paper but are
; our versions of classical theorems.
```

THEOREM: always-in-particular

```
((z ∈ a)
 ∧   litatom (v)
 ∧   for (v,
          a,
          ''t,
          'always,
          x,
          al))
→    eval$ (t, x, cons (cons (v, z), al))
```

THEOREM: exists-in-particular

```
((z ∈ a) ∧ eval$ (t, x, cons (cons (v, z), al)) ∧ litatom (v))
→    for (v,
          a,
          ''t,
          'exists,
          x,
          al)
```

THEOREM: exists-vs-always

for (v,
    a,
    c,
    `'always`,
    x,
    al)
= (¬ for (v,
        a,
        c,
        `'exists`,
        list (`'not`, x),
        al))

```
; Now I move on to the simple theorems about FOLDR.
```

Definition:
foldr (op, k, lst)
= **if** $lst \simeq$ **nil** **then** k
  else apply\$ (op, list (car (lst), foldr (op, k, cdr (lst)))) **endif**

Theorem: foldr-append
foldr (op, k, append (a, b)) = foldr (op, foldr (op, k, b), a)

Theorem: foldr-cons-is-append
foldr (`'cons`, b, a) = append (a, b)

Theorem: foldr-and-t-is-for-always
foldr (`'and`, **t**, lst)
= **for** p **in** lst
  always p **endfor**

Theorem: foldr-or-f-is-for-exists
foldr (`'or`, **f**, lst)
= **for** p **in** lst
  exists p **endfor**

Theorem: foldr-plus-0-is-for-sum
foldr (`'plus`, `0`, lst)
= **for** x **in** lst
  sum x **endfor**

Theorem: foldr-times-1-is-for-multiply
foldr (`'times`, `1`, lst)
= **for** x **in** lst
  multiply x **endfor**

```
; Section 10.  Proof of the Binomial Theorem

; In the proof we cite a more general quantifier manipulation
; lemma than we actually used in the binomial proof.  I prove
; it here.
```

DEFINITION:
from-to(i, j)
$=$ **if** $j < i$ **then nil**
elseif fix$(i) =$ fix(j) **then** list(fix(j))
else append(from-to$(i, j - 1)$, list(j)) **endif**

THEOREM: eval\$-substitute
litatom(v)
$\rightarrow$ (eval\$(*flg*, substitute(*form*, v, *flg*, *body*), *alist*)
$=$ eval\$(*flg*, *body*, cons(cons(v, eval\$(**t**, *form*, *alist*)), *alist*)))

THEOREM: redundant-binding-can-be-deleted
eval\$(*flg*, x, cons(cons(v, *val1*), cons(cons(v, *val2*), *alist*)))
$=$ eval\$(*flg*, x, cons(cons(v, *val1*), *alist*))

DEFINITION:
down-shift(l)
$=$ **if** $l \simeq$ **nil then nil**
else cons(car$(l) - 1$, down-shift(cdr(l))) **endif**

THEOREM: down-shift-from-to
$((j \not\simeq 0) \wedge (k \not\simeq 0))$
$\rightarrow$ (down-shift(from-to(j, k)) $=$ from-to$(j - 1, k - 1)$)

DEFINITION:
all-non-zerop(l)
$=$ **if** listp(l) **then** (car$(l) \not\simeq 0$) $\wedge$ all-non-zerop(cdr(l))
else t endif

THEOREM: all-non-zerop-append
all-non-zerop(append(a, b)) $=$ (all-non-zerop(a) $\wedge$ all-non-zerop(b))

THEOREM: all-non-zerop-from-to
$(j \not\simeq 0) \rightarrow$ all-non-zerop(from-to(j, k))

THEOREM: down-shift-for
(litatom(v) $\wedge$ all-non-zerop(l))
$\rightarrow$ (for$(v$,
down-shift(l),

```
        substitute (list ('add1, v), v, t, cond),
        op,
        substitute (list ('add1, v), v, t, body),
        alist)
    =   for (v,
             l,
             cond,
             op,
             body,
             alist))
```

```
; !!! BUG IN THE PAPER !!!
; The range shift theorem was wrong!  I needed to add that K also was
; not ZEROP.  If J=1 and K=0 then (FROM-TO J K) is NIL but
; (FROM-TO (SUB1 J) (SUB1 K)) is (FROM-TO 0 0) = '(0).
```

Theorem: range-shift

```
 (litatom (v) ∧ (j ≄ 0) ∧ (k ≄ 0))
 →  (for (v,
        from-to (j, k),
        cond,
        op,
        body,
        alist)
    =   for (v,
             from-to (j − 1, k − 1),
             substitute (list ('add1, v), v, t, cond),
             op,
             substitute (list ('add1, v), v, t, body),
             alist))
```

Index