```
#|
 Copyright (C) 1994 by Natarajan Shankar.  All Rights Reserved.

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

 NO WARRANTY

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

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

|#
```

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

```
;An Annotated Script of A MECHANICAL PROOF OF THE CHURCH-ROSSER THEOREM.
;                          by
;                    N. Shankar
;
;What follows is a list of events (definitions and lemmas) in the mechanical
;proof of the Church-Rosser theorem that was carried out with the Boyer-Moore
;theorem-prover.  The formal expressions below are subtle and complex and
;the annotations provide only a small amount of help.  The proof consists of
;the following parts -
;1. Formalization of Lambda-calculus in the standard notation.
;2. Formalization of Lambda-calculus in the de Bruijn notation.
;3. Proof of the Diamond property of walks in the de Bruijn notation.
;4. Proof of the Diamond property of sequences of beta-steps.
;5. Proof that a sequence of beta-steps is the transitive closure of
;   a walk.
;6. Proof that the standard notation translates into the de Bruijn notation.
;7. Proof that the Church-Rosser proof for the de Bruijn notation can be
```

```
;   translated back into the standard notation.
;8. Proof of the Diamond property of walks directly in the standard notation.
;standard formalization:  The prefix "N" stands for "normal".  The two shells
;NLAMBDA and NCOMB represent lambda-abstraction and application, respectively.
;Variables are represented by numbers and constants by literal atoms.
```

EVENT: Add the shell *nlambda*, with recognizer function symbol *nlambdap* and 2 accessors: *nbind*, with type restriction (one-of numberp) and default value zero; *nbody*, with type restriction (none-of) and default value zero.

EVENT: Add the shell *ncomb*, with recognizer function symbol *ncombp* and 2 accessors: *nleft*, with type restriction (none-of) and default value zero; *nright*, with type restriction (none-of) and default value zero.

```
;(NSUBST X Y N) is the result of substituting term Y for variable N in term X.
```

DEFINITION:
nsubst $(x,\ y,\ n)$
= **if** nlambdap (x)
  then if nbind $(x) = n$ **then** x
    else nlambda (nbind (x), nsubst (nbody (x), y, n)) **endif**
  elseif ncombp (x)
  then ncomb (nsubst (nleft (x), y, n), nsubst (nright (x), y, n))
  elseif $x \in \mathbf{N}$
  then if $x = n$ **then** y
    else x **endif**
  else x **endif**

```
;Variable X does not occur free in term Y.
```

DEFINITION:
not-free-in $(x,\ y)$
= **if** nlambdap (y)
  then if $x =$ nbind (y) **then t**
    else not-free-in (x, nbody (y)) **endif**
  elseif ncombp (y)
  then not-free-in (x, nleft (y)) $\land$ not-free-in (x, nright (y))
  else $x \neq y$ **endif**

```
;The free variables in Y are disjoint from the bound variables in X.
```

DEFINITION:
free-for $(x,\ y)$
= **if** nlambdap (x) **then** not-free-in (nbind (x), y)

$\wedge$ free-for (nbody (x), y)
elseif ncombp (x) **then** free-for (nleft (x), y)
$\wedge$ free-for (nright (x), y)
else t endif

```
;Index of the first occurrence of N in the list LIST.
```

DEFINITION:
index $(n,\ list)$
= **if** listp $(list)$
then if car $(list) = n$ **then** `1`
else $1 +$ index $(n$, cdr $(list))$ **endif**
else $1 + n$ **endif**

```
;A and B are identical modulo bound-variables which are accumulated in
;X and Y, respectively.
```

DEFINITION:
alpha-equal $(a,\ b,\ x,\ y)$
= **if** nlambdap $(a) \wedge$ nlambdap (b)
then alpha-equal (nbody (a), nbody (b), cons (nbind (a), x), cons (nbind (b), y))
elseif ncombp $(a) \wedge$ ncombp (b)
then alpha-equal (nleft (a), nleft (b), x, y)
$\wedge$ alpha-equal (nright (a), nright (b), x, y)
elseif $(a \in \mathbf{N}) \wedge (b \in \mathbf{N})$ **then** index $(a,\ x) =$ index $(b,\ y)$
else $a = b$ **endif**

```
;X is a well-formed term in the standard notation.
```

DEFINITION:
ntermp (x)
= **if** nlambdap (x) **then** ntermp (nbody (x))
elseif ncombp (x) **then** ntermp (nleft (x)) $\wedge$ ntermp (nright (x))
else $(x \in \mathbf{N}) \vee$ litatom (x) **endif**

```
;A goes to B by the beta-reduction of some non-overlapping redexes.
```

DEFINITION:
nbeta-step $(a,\ b)$
= **if** $a = b$ **then t**
elseif nlambdap (a)
then nlambdap (b)
$\wedge$ (nbind $(a) =$ nbind (b))
$\wedge$ nbeta-step (nbody (a), nbody (b))
elseif ncombp (a)

then (nlambdap (nleft (a))
$\quad\wedge\quad$ free-for (nbody (nleft (a)), nright (a))
$\quad\wedge\quad$ (b = nsubst (nbody (nleft (a)), nright (a), nbind (nleft (a)))))
$\vee\quad$ (ncombp (b)
$\quad\wedge\quad$ nbeta-step (nleft (a), nleft (b))
$\quad\wedge\quad$ nbeta-step (nright (a), nright (b)))
else f endif

```
;A goes to B by an alpha-or-beta-step.
```

DEFINITION:
nstep (a, b) = (alpha-equal (a, b, **nil**, **nil**) $\vee$ nbeta-step (a, b))

```
;A goes to B via a series of steps through terms in LIST.
```

DEFINITION:
nreduction (a, b, *list*)
= **if** listp (*list*)
then nstep (car (*list*), b) $\wedge$ nreduction (a, car (*list*), cdr (*list*))
else nstep (a, b) **endif**

```
;de Bruijn formalization:
```

EVENT: Add the shell *lambda*, with recognizer function symbol *lambdap* and 1 accessor: *body*, with type restriction (none-of) and default value zero.

EVENT: Add the shell *comb*, with recognizer function symbol *combp* and 2 accessors: *left*, with type restriction (none-of) and default value zero; *right*, with type restriction (none-of) and default value zero.

```
;BUMP increments all variables of stack-level greater than N by one.
```

DEFINITION:
bump (x, n)
= **if** lambdap (x) **then** lambda (bump (body (x), $1 + n$))
elseif combp (x) **then** comb (bump (left (x), n), bump (right (x), n))
elseif $n < x$ **then** $1 + x$
else x **endif**

```
;(SUBST X Y N) is the result of substituting Y for N in X.
```

DEFINITION:
subst (x, y, n)
= **if** lambdap (x) **then** lambda (subst (body (x), bump (y, 0), $1 + n$))
elseif combp (x) **then** comb (subst (left (x), y, n), subst (right (x), y, n))

elseif $x \not\simeq \texttt{0}$
then if $x = n$ **then** y
elseif $n < x$ **then** $x - 1$
else x **endif**
else x **endif**

```
;BUMP commutes with itself.
```

THEOREM: bump-bump
$(n \leq m) \rightarrow (\text{bump}\,(\text{bump}\,(y,\ n),\ 1 + m) = \text{bump}\,(\text{bump}\,(y,\ m),\ n))$

DEFINITION:
bump-subst-induct $(x,\ y,\ n,\ m)$
$=$ **if** lambdap (x) **then** bump-subst-induct (body (x), bump $(y,\ \texttt{0})$, $1 + n$, $1 + m$)
elseif combp (x)
then bump-subst-induct (left (x), y, n, m)
$\wedge$ bump-subst-induct (right (x), y, n, m)
else t endif

```
;BUMP distributes over SUBST.
```

THEOREM: bump-subst
$(m < n) \rightarrow (\text{bump}\,(\text{subst}\,(x,\ y,\ n),\ m) = \text{subst}\,(\text{bump}\,(x,\ m),\ \text{bump}\,(y,\ m),\ 1 + n))$

DEFINITION:
subst-induct $(x,\ y,\ z,\ m,\ n)$
$=$ **if** lambdap (x)
then subst-induct (body (x), bump $(y,\ \texttt{0})$, bump $(z,\ \texttt{0})$, $1 + m$, $1 + n$)
elseif combp (x)
then subst-induct (left (x), y, z, m, n)
$\wedge$ subst-induct (right (x), y, z, m, n)
else t endif

THEOREM: lambdap-bump
$\text{lambdap}\,(x) \rightarrow \text{lambdap}\,(\text{bump}\,(x,\ n))$

THEOREM: lambdap-subst
$\text{lambdap}\,(x) \rightarrow \text{lambdap}\,(\text{subst}\,(x,\ y,\ n))$

```
;BUMP distributes over SUBST in another way.
```

THEOREM: another-bump-subst
$(n \leq (1 + m))$
$\rightarrow (\text{subst}\,(\text{bump}\,(x,\ 1 + m),\ \text{bump}\,(y,\ m),\ n) = \text{bump}\,(\text{subst}\,(x,\ y,\ n),\ m))$

THEOREM: bump-lambdap
$\text{lambdap}\,(x) \rightarrow (\text{body}\,(\text{bump}\,(x,\ n)) = \text{bump}\,(\text{body}\,(x),\ 1 + n))$

THEOREM: subst-lambda
lambdap $(x) \rightarrow (\text{body}(\text{subst}(x, y, n)) = \text{subst}(\text{body}(x), \text{bump}(y, \texttt{0}), 1 + n))$

DEFINITION: left-instrs $(x) = \text{car}(x)$

DEFINITION: right-instrs $(x) = \text{cadr}(x)$

DEFINITION: command $(x) = \text{caddr}(x)$

`;Result of applying walk-instruction W to M in an inside-out manner.`

DEFINITION:
walk (w, m)
= **if** lambdap (m) **then** lambda (walk $(w$, body $(m))$)
  elseif combp (m)
  then if (command (w) = `'reduce`) $\wedge$ lambdap (left (m))
    then subst (body (walk (left-instrs (w), left (m))),
        walk (right-instrs (w), right (m)),
        `1`)
    else comb (walk (left-instrs (w), left (m)),
        walk (right-instrs (w), right (m))) **endif**
  else m **endif**

DEFINITION:
bump-walk-ind (u, m, n)
= **if** lambdap (m) **then** bump-walk-ind $(u$, body (m), $1 + n)$
  elseif combp (m)
  then if (command (u) = `'reduce`) $\wedge$ lambdap (left (m))
    then bump-walk-ind (left-instrs (u), left (m), n)
      $\wedge$ bump-walk-ind (right-instrs (u), right (m), n)
    else bump-walk-ind (left-instrs (u), left (m), n)
      $\wedge$ bump-walk-ind (right-instrs (u), right (m), n) **endif**
  else t endif

THEOREM: lambda-walk
lambdap $(m) \rightarrow$ lambdap (walk (w, m))

`;BUMP and WALK commute with each other.`

THEOREM: bump-walk
walk $(u$, bump $(m, n))$ = bump (walk (u, m), n)

THEOREM: nlistp-walk
$(w \simeq \textbf{nil}) \rightarrow (\text{walk}(w, m) = m)$

`;The 'skolem' function generating the substitutive walk.`

DEFINITION:
sub-walk (w, m, u, n)
$=$ **if** lambdap (m) **then** sub-walk $(w, \text{body}(m), u, 1 + n)$
elseif combp (m)
then if $(\text{command}(w) = $ `'reduce`$) \land$ lambdap (left (m))
then list (sub-walk (left-instrs (w), left (m), u, n),
sub-walk (right-instrs (w), right (m), u, n),
`'reduce`)
else list (sub-walk (left-instrs (w), left (m), u, n),
sub-walk (right-instrs (w), right (m), u, n)) **endif**
elseif $m \not\simeq$ `0`
then if $m = n$ **then** u
else w **endif**
else w **endif**

DEFINITION:
walk-subst-ind (w, m, u, x, n)
$=$ **if** lambdap (m) **then** walk-subst-ind $(w, \text{body}(m), u, \text{bump}(x, 0), 1 + n)$
elseif combp (m)
then if $(\text{command}(w) = $ `'reduce`$) \land$ lambdap (left (m))
then walk-subst-ind (left-instrs (w), left (m), u, x, n)
$\land$ walk-subst-ind (right-instrs (w), right (m), u, x, n)
else walk-subst-ind (left-instrs (w), left (m), u, x, n)
$\land$ walk-subst-ind (right-instrs (w), right (m), u, x, n) **endif**
else t endif

```
;(BUMP X N) cannot contain N+1 free.
```

THEOREM: subst-not-free-in
$\text{subst}(\text{bump}(x, n), y, 1 + n) = x$

```
;SUBST distributes over itself.  A tricky one!
```

THEOREM: subst-subst
$((m \in \mathbf{N}) \land (m < n))$
$\rightarrow$ $(\text{subst}(\text{subst}(x, \text{bump}(z, m), 1 + n), \text{subst}(y, z, n), 1 + m)$
$=$ $\text{subst}(\text{subst}(x, y, 1 + m), z, n))$

```
;The substitutivity of walks lemma.
```

THEOREM: walk-subst
$(n \not\simeq 0)$
$\rightarrow$ $(\text{walk}(\text{sub-walk}(w, m, u, n), \text{subst}(m, x, n))$
$=$ $\text{subst}(\text{walk}(w, m), \text{walk}(u, x), n))$

THEOREM: walk-lambda
lambdap $(x) \rightarrow$ (body (walk (u, x)) $=$ walk $(u$, body $(x)))$

```
;‘Skolem’ function generating convergent walk.
```

DEFINITION:
make-walk (m, u, v)
$=$ **if** lambdap (m) **then** make-walk (body (m), u, v)
elseif combp (m)
then if (command (u) $=$ `'reduce`) $\wedge$ lambdap (left (m))
then sub-walk (make-walk (left (m), left-instrs (u), left-instrs (v)),
body (walk (left-instrs (u), left (m))),
make-walk (right (m), right-instrs (u), right-instrs (v)),
`1`)
elseif (command (v) $=$ `'reduce`) $\wedge$ lambdap (left (m))
then list (make-walk (left (m), left-instrs (u), left-instrs (v)),
make-walk (right (m), right-instrs (u), right-instrs (v)),
`'reduce`)
else list (make-walk (left (m), left-instrs (u), left-instrs (v)),
make-walk (right (m),
right-instrs (u),
right-instrs (v))) **endif**
else u **endif**

```
;The Diamond property of walks.
```

THEOREM: main
walk (make-walk (m, u, v), walk (u, m)) $=$ walk (make-walk (m, v, u), walk (v, m))

```
;Result of applying a series of walks.
```

DEFINITION:
reduce (w, m)
$=$ **if** listp (w) **then** reduce (cdr (w), walk (car (w), m))
else m **endif**

```
;‘Skolem’ function constructing convergent walk given a walk W and a
;series of walks V.
```

DEFINITION:
make-walk-reduce (m, w, v)
$=$ **if** listp (v)
then make-walk-reduce (walk (car (v), m), make-walk (m, car (v), w), cdr (v))
else w **endif**

```
;‘Skolem’ function constructing convergent series of walks for same case as above.
```

DEFINITION:
make-reduce-walk $(m,\ w,\ v)$

```
=   if listp (v)
    then cons (make-walk (m, w, car (v)),
               make-reduce-walk (walk (car (v), m),
                                 make-walk (m, car (v), w),
                                 cdr (v)))
    else nil endif
```

;The Rectangle property.

THEOREM: walk-reduce
reduce (make-reduce-walk $(m,\ w,\ v)$, walk $(w,\ m)$)
= walk (make-walk-reduce $(m,\ w,\ v)$, reduce $(v,\ m)$)

;'Skolem' function constructing convergent series of walks for Diamond property.

DEFINITION:
make-reduce $(m,\ u,\ v)$

```
=   if listp (u)
    then cons (make-walk-reduce (m, car (u), v),
               make-reduce (walk (car (u), m),
                            cdr (u),
                            make-reduce-walk (m, car (u), v)))
    else u endif
```

THEOREM: list-make-reduce

```
listp (v)
→  (reduce (make-reduce (m, u, v), reduce (v, m))
      =  reduce (make-reduce (walk (car (v), m),
                              make-reduce-walk (m, car (v), u),
                              cdr (v)),
                 reduce (v, m)))
```

THEOREM: list-make-reduce1
reduce (make-reduce $(m,\ u,$ cons $(x,\ y))$, reduce $(y,$ walk $(x,\ m)))$
= reduce (make-reduce (walk $(x,\ m)$, make-reduce-walk $(m,\ x,\ u),\ y)$,
reduce (cons $(x,\ y),\ m))$

;The Diamond property of a series of walks.

THEOREM: church-rosser
reduce (make-reduce $(m,\ u,\ v)$, reduce $(v,\ m)$)
= reduce (make-reduce $(m,\ v,\ u)$, reduce $(u,\ m)$)

;Translation from standard to de Bruijn. X a standard term, and
;BOUNDS bound variables accumulated so far.

DEFINITION:
translate $(x,\ bounds)$
$=$ **if** nlambdap (x) **then** lambda (translate (nbody (x), cons (nbind (x), $bounds$)))
elseif ncombp (x)
then comb (translate (nleft (x), $bounds$), translate (nright (x), $bounds$))
elseif $x \in \mathbf{N}$ **then** index $(x,\ bounds)$
else x **endif**

```
;Two alpha-equivalent terms have the same translation.
```

THEOREM: alpha-translate
alpha-equal $(a,\ b,\ x,\ y) \rightarrow$ (translate $(a,\ x)$ $=$ translate $(b,\ y)$)

DEFINITION:
insert $(x,\ bounds,\ n)$
$=$ **if** $n \simeq 0$ **then** cons $(x,\ bounds)$
elseif listp $(bounds)$
then cons (car $(bounds)$, insert $(x$, cdr $(bounds),\ n - 1))$
else cons $(x$, **nil**$)$ **endif**

DEFINITION:
nsubst-ind $(x,\ y,\ z,\ bounds,\ n)$
$=$ **if** nlambdap (x)
then if nbind $(x) = z$ **then t**
else nsubst-ind (nbody (x), y, z, cons (nbind (x), $bounds$), $1 + n$) **endif**
elseif ncombp (x)
then nsubst-ind (nleft (x), y, z, $bounds$, n)
$\wedge$ nsubst-ind (nright (x), y, z, $bounds$, n)
else t endif

DEFINITION:
trans-insert-ind $(x,\ z,\ bounds,\ n)$
$=$ **if** nlambdap (x)
then trans-insert-ind (nbody (x), z, cons (nbind (x), $bounds$), $1 + n$)
elseif ncombp (x)
then trans-insert-ind (nleft (x), z, $bounds$, n)
$\wedge$ trans-insert-ind (nright (x), z, $bounds$, n)
else t endif

DEFINITION:
precede $(x,\ bounds,\ n)$
$=$ **if** $n \simeq 0$ **then f**
elseif listp $(bounds)$
then (car $(bounds) = x$) $\vee$ precede $(x$, cdr $(bounds),\ n - 1)$
else f endif

THEOREM: nprecede-index
$((\neg\ \text{precede}(z,\ \mathit{bounds},\ n)) \land (n \leq \text{length}(\mathit{bounds})))$
$\rightarrow\quad (\text{index}(z,\ \text{insert}(z,\ \mathit{bounds},\ n)) = (1 + n))$

THEOREM: precede-index1
$((n < \text{index}(x,\ \mathit{bounds})) \land (n \leq \text{length}(\mathit{bounds})))$
$\rightarrow\quad (\text{index}(x,\ \text{insert}(z,\ \mathit{bounds},\ n))$
$\qquad = \ \ \textbf{if}\ x = z\ \ \textbf{then}\ 1 + n$
$\qquad\qquad \textbf{else}\ 1 + \text{index}(x,\ \mathit{bounds})\ \textbf{endif})$

THEOREM: precede-index
$\text{precede}(z,\ \mathit{bounds},\ n) \rightarrow (\text{index}(z,\ \text{insert}(z,\ \mathit{bounds},\ n)) < (1 + n))$

THEOREM: precede-index2
$((\text{index}(x,\ \mathit{bounds}) \leq n) \land (n \leq \text{length}(\mathit{bounds})))$
$\rightarrow\quad (\text{index}(x,\ \text{insert}(z,\ \mathit{bounds},\ n)) = \text{index}(x,\ \mathit{bounds}))$

THEOREM: translate-insert
$(\text{precede}(z,\ \mathit{bounds},\ n) \land (n \leq \text{length}(\mathit{bounds})) \land \text{ntermp}(x))$
$\rightarrow\quad (\text{translate}(x,\ \text{insert}(z,\ \mathit{bounds},\ n)) = \text{bump}(\text{translate}(x,\ \mathit{bounds}),\ n))$

THEOREM: subst-nsubst
$(\text{precede}(z,\ \mathit{bounds},\ n) \land \text{ntermp}(x) \land (n \leq \text{length}(\mathit{bounds})))$
$\rightarrow\quad (\text{subst}(\text{translate}(x,\ \text{insert}(z,\ \mathit{bounds},\ n)),\ y,\ 1 + n)$
$\qquad = \ \ \text{translate}(x,\ \mathit{bounds}))$

DEFINITION:
$\text{not-free-ind}(y,\ z,\ \mathit{bounds},\ n)$
$= \ \ \textbf{if}\ \text{nlambdap}(y)$
$\quad \textbf{then}\ \text{not-free-ind}(\text{nbody}(y),\ z,\ \text{cons}(\text{nbind}(y),\ \mathit{bounds}),\ 1 + n)$
$\quad \textbf{elseif}\ \text{ncombp}(y)$
$\quad \textbf{then}\ \text{not-free-ind}(\text{nleft}(y),\ z,\ \mathit{bounds},\ n)$
$\qquad\quad \land \ \ \text{not-free-ind}(\text{nright}(y),\ z,\ \mathit{bounds},\ n)$
$\quad \textbf{else}\ \textbf{t}\ \textbf{endif}$

THEOREM: translate-insert1
$(\text{precede}(z,\ \text{cons}(x,\ \mathit{bounds}),\ 1 + n) \land \text{ntermp}(y) \land (n \leq \text{length}(\mathit{bounds})))$
$\rightarrow\quad (\text{translate}(y,\ \text{cons}(x,\ \text{insert}(z,\ \mathit{bounds},\ n)))$
$\qquad = \ \ \text{bump}(\text{translate}(y,\ \text{cons}(x,\ \mathit{bounds})),\ 1 + n))$

THEOREM: not-free-in-translate
$(\text{not-free-in}(z,\ y) \land (n \leq \text{length}(\mathit{bounds})) \land \text{ntermp}(y))$
$\rightarrow\quad (\text{translate}(y,\ \text{insert}(z,\ \mathit{bounds},\ n)) = \text{bump}(\text{translate}(y,\ \mathit{bounds}),\ n))$

THEOREM: not-free-in-corr
$(\text{ntermp}(y) \land \text{not-free-in}(z,\ y))$
$\rightarrow\quad (\text{translate}(y,\ \text{cons}(z,\ \mathit{bounds})) = \text{bump}(\text{translate}(y,\ \mathit{bounds}),\ \mathtt{0}))$

THEOREM: subst-nsubst2
$(\text{precede}(z,\ \text{cons}(w,\ \mathit{bounds}),\ 1 + n) \land \text{ntermp}(x) \land (n \leq \text{length}(\mathit{bounds})))$
$\rightarrow\ (\text{subst}(\text{translate}(x,\ \text{cons}(w,\ \text{insert}(z,\ \mathit{bounds},\ n))),\ y,\ 1 + (1 + n))$
$\quad = \ \text{translate}(x,\ \text{cons}(w,\ \mathit{bounds})))$

THEOREM: subst-nsubst2-corr
$\text{ntermp}(x)$
$\rightarrow\ (\text{subst}(\text{translate}(x,\ \text{cons}(z,\ \text{cons}(z,\ \mathit{bounds}))),\ y,\ 1 + (1 + \mathtt{0}))$
$\quad = \ \text{translate}(x,\ \text{cons}(z,\ \mathit{bounds})))$

THEOREM: index-insert
$((x \neq z) \land (n \leq \text{length}(\mathit{bounds})))$
$\rightarrow\ (\text{index}(x,\ \text{insert}(z,\ \mathit{bounds},\ n)) \neq (1 + n))$

```
;Translation distributes over substitution.
```

THEOREM: translate-preserves-subst
$(\text{free-for}(x,\ y)$
$\land\ (\neg\ \text{precede}(z,\ \mathit{bounds},\ n))$
$\land\ (n \leq \text{length}(\mathit{bounds}))$
$\land\ \text{ntermp}(x)$
$\land\ \text{ntermp}(y))$
$\rightarrow\ (\text{translate}(\text{nsubst}(x,\ y,\ z),\ \mathit{bounds})$
$\quad = \ \text{subst}(\text{translate}(x,\ \text{insert}(z,\ \mathit{bounds},\ n)),$
$\qquad\qquad \text{translate}(y,\ \mathit{bounds}),$
$\qquad\qquad 1 + n))$

```
;Translation distributes over beta-reduction.
```

THEOREM: translate-preserves-reduction
$(\text{nlambdap}(x) \land \text{free-for}(\text{nbody}(x),\ y) \land \text{ntermp}(x) \land \text{ntermp}(y))$
$\rightarrow\ (\text{translate}(\text{nsubst}(\text{nbody}(x),\ y,\ \text{nbind}(x)),\ \mathit{bounds})$
$\quad = \ \text{subst}(\text{body}(\text{translate}(x,\ \mathit{bounds})),\ \text{translate}(y,\ \mathit{bounds}),\ \mathtt{1}))$

```
;Reduction is the transitive closure of a walk.
;X goes to Y in a beta-step, in the de Bruijn notation.
```

DEFINITION:
$\text{beta-step}(x,\ y)$
$=\ $ **if** $x = y$ **then t**
elseif $\text{lambdap}(x)$ **then** $\text{lambdap}(y) \land \text{beta-step}(\text{body}(x),\ \text{body}(y))$
elseif $\text{combp}(x)$
then $(\text{lambdap}(\text{left}(x)) \land (y = \text{subst}(\text{body}(\text{left}(x)),\ \text{right}(x),\ \mathtt{1})))$
$\quad \lor\ (\text{combp}(y)$
$\qquad \land\ \text{beta-step}(\text{left}(x),\ \text{left}(y))$
$\qquad \land\ \text{beta-step}(\text{right}(x),\ \text{right}(y)))$
else f endif

```
;A goes to B via LIST in a series of beta-steps.
```

DEFINITION:
reduction $(a,\ b,\ \mathit{list})$
$=$ **if** listp (list)
then beta-step (car (list), b) $\wedge$ reduction $(a$, car (list), cdr $(\mathit{list}))$
else beta-step $(a,\ b)$ **endif**

DEFINITION:
make-walk-step $(x,\ y)$
$=$ **if** $x = y$ **then nil**
elseif lambdap (x) **then** make-walk-step (body (x), body (y))
elseif combp (x)
then if lambdap (left (x))
$\wedge$ (y = subst (body (left (x)), right (x), `1`))
then `'(nil nil reduce)`
else list (make-walk-step (left (x), left (y)),
make-walk-step (right (x), right (y))) **endif**
else nil endif

THEOREM: make-walk-step-walks
beta-step $(a,\ b) \rightarrow$ (walk (make-walk-step $(a,\ b)$, a) $= b$)

DEFINITION:
append $(x,\ y)$
$=$ **if** listp (x) **then** cons (car (x), append (cdr (x), y))
else y **endif**

DEFINITION:
make-reduce-reduction $(a,\ b,\ \mathit{list})$
$=$ **if** listp (list)
then append (make-reduce-reduction (a, car (list), cdr (list)),
list (make-walk-step (car (list), b), **nil**))
else list (make-walk-step $(a,\ b)$) **endif**

THEOREM: reduce-append
reduce (append $(x,\ y)$, a) $=$ reduce (y, reduce $(x,\ a)$)

```
;A reduction is a series of walks.
```

THEOREM: make-reduce-reduction-reduction
reduction $(a,\ b,\ \mathit{list}) \rightarrow$ (reduce (make-reduce-reduction $(a,\ b,\ \mathit{list})$, a) $= b$)

DEFINITION:
list-lambda (list)
$=$ **if** listp (list) **then** cons (lambda (car (list)), list-lambda (cdr (list)))
else nil endif

THEOREM: lambda-reduction
reduction (a, $a1$, $list$) → reduction (lambda (a), lambda ($a1$), list-lambda ($list$))

DEFINITION:
map-comb-left (a, $list$)
= **if** listp ($list$)
then cons (comb (a, car ($list$)), map-comb-left (a, cdr ($list$)))
else nil endif

DEFINITION:
map-comb-right (a, $list$)
= **if** listp ($list$)
then cons (comb (car ($list$), a), map-comb-right (a, cdr ($list$)))
else nil endif

DEFINITION:
list-comb (a, b, $list1$, $list2$)
= **if** listp ($list1$)
then if listp ($list2$)
then cons (comb (car ($list1$), car ($list2$)),
list-comb (a, b, cdr ($list1$), cdr ($list2$)))
else map-comb-right (b, $list1$) **endif**
elseif listp ($list2$) **then** map-comb-left (a, $list2$)
else nil endif

THEOREM: map-comb-right-reduction2
reduction (a, $a1$, $list1$)
→ reduction (comb (a, $b1$), comb ($a1$, $b1$), map-comb-right ($b1$, $list1$))

THEOREM: map-comb-right-reduction
(beta-step (b, $b1$) ∧ reduction (a, $a1$, $list1$))
→ reduction (comb (a, b), comb ($a1$, $b1$), map-comb-right (b, $list1$))

THEOREM: map-comb-left-reduction2
reduction (b, $b1$, $list2$)
→ reduction (comb ($a1$, b), comb ($a1$, $b1$), map-comb-left ($a1$, $list2$))

THEOREM: map-comb-left-reduction
(reduction (b, $b1$, $list2$) ∧ beta-step (a, $a1$))
→ reduction (comb (a, b), comb ($a1$, $b1$), map-comb-left (a, $list2$))

THEOREM: list-comb-reduction
(reduction (a, $a1$, $list1$) ∧ reduction (b, $b1$, $list2$))
→ reduction (comb (a, b), comb ($a1$, $b1$), list-comb (a, b, $list1$, $list2$))

THEOREM: beta-subst
beta-step (comb (lambda (a), b), subst (a, b, `1`))

```
DEFINITION:
make-reduction-walk (w, a)
=    if walk (w, a) = a  then nil
     elseif lambdap (a)  then list-lambda (make-reduction-walk (w, body (a)))
     elseif combp (a)
     then if lambdap (left (a)) ∧ (command (w) = 'reduce)
          then cons (subst (body (walk (left-instrs (w), left (a))),
                            walk (right-instrs (w), right (a)),
                            1),
                     cons (comb (walk (left-instrs (w), left (a)),
                                 walk (right-instrs (w), right (a))),
                           list-comb (left (a),
                                      right (a),
                                      make-reduction-walk (left-instrs (w),
                                                           left (a)),
                                      make-reduction-walk (right-instrs (w),
                                                           right (a)))))
          else list-comb (left (a),
                          right (a),
                          make-reduction-walk (left-instrs (w), left (a)),
                          make-reduction-walk (right-instrs (w),
                                               right (a))) endif
     else nil endif

;A walk is a series of beta-steps.

THEOREM: make-reduction-walk-steps
 reduction (a, walk (w, a), make-reduction-walk (w, a))

THEOREM: reduction-append
 (reduction (a, b, x) ∧ reduction (b, c, y))
 →    reduction (a, c, append (y, cons (b, x)))

DEFINITION:
make-reduce-steps (w, a)
=    if listp (w)
     then append (make-reduce-steps (cdr (w), walk (car (w), a)),
                  cons (walk (car (w), a), make-reduction-walk (car (w), a)))
     else nil endif

THEOREM: make-reduce-steps-steps
 reduction (a, reduce (w, a), make-reduce-steps (w, a))

;'Skolem' function MAKE-W constructs the W to which Y and Z converge.
```

DEFINITION:
make-w $(x,\ y,\ z,\ \mathit{list1},\ \mathit{list2})$
$=$ reduce (make-reduce $(x,$
make-reduce-reduction $(x,\ y,\ \mathit{list1})$,
make-reduce-reduction $(x,\ z,\ \mathit{list2})$),
reduce (make-reduce-reduction $(x,\ z,\ \mathit{list2}),\ x)$)

```
;MAKE-REDUCTION constructs the corresponding series of beta-steps.
```

DEFINITION:
make-reduction $(x,\ y,\ z,\ \mathit{list1},\ \mathit{list2})$
$=$ make-reduce-steps (make-reduce $(x,$
make-reduce-reduction $(x,\ z,\ \mathit{list2})$,
make-reduce-reduction $(x,\ y,\ \mathit{list1})$),
$y)$

```
;The next two events together give the Church-Rosser theorem for the de Bruijn
;representation.  The first one shows the existence of convergent reductions,
;and the second one shows that they converge.
```

THEOREM: the-real-church-rosser
(reduction $(x,\ y,\ \mathit{list1})$ $\wedge$ reduction $(x,\ z,\ \mathit{list2})$)
$\rightarrow$ (reduction $(y,$
make-w $(x,\ z,\ y,\ \mathit{list2},\ \mathit{list1})$,
make-reduction $(x,\ y,\ z,\ \mathit{list1},\ \mathit{list2})$)
$\wedge$ reduction $(z,$
make-w $(x,\ y,\ z,\ \mathit{list1},\ \mathit{list2})$,
make-reduction $(x,\ z,\ y,\ \mathit{list2},\ \mathit{list1})$))

THEOREM: both-make-w-are-same
(reduction $(x,\ y,\ \mathit{list1})$ $\wedge$ reduction $(x,\ z,\ \mathit{list2})$)
$\rightarrow$ (make-w $(x,\ y,\ z,\ \mathit{list1},\ \mathit{list2})$ $=$ make-w $(x,\ z,\ y,\ \mathit{list2},\ \mathit{list1})$)

```
;Translating the proof back into the standard notation.
;A standard beta-step translates to a de Bruijn beta-step.
```

THEOREM: nbeta-step-translates
(ntermp (a) $\wedge$ nbeta-step $(a,\ b)$)
$\rightarrow$ beta-step (translate $(a,\ x)$, translate $(b,\ x)$)

DEFINITION:
trans-list (x)
$=$ **if** listp (x) **then** cons (translate (car (x), **nil**), trans-list (cdr (x)))
else nil endif

THEOREM: ntermp-subst
(ntermp (x) $\wedge$ ntermp (y)) $\rightarrow$ ntermp (nsubst $(x,\ y,\ n)$)

THEOREM: ntermp-bstep
$(\text{nbeta-step}(a, b) \wedge \text{ntermp}(a)) \rightarrow \text{ntermp}(b)$

THEOREM: ntermp-astep
$(\text{alpha-equal}(a, b, x, y) \wedge \text{ntermp}(a)) \rightarrow \text{ntermp}(b)$

THEOREM: ntermp-list
$(\text{nreduction}(a, b, \mathit{list}) \wedge \text{ntermp}(a)) \rightarrow \text{ntermp}(b)$

```
;Standard reduction goes to de Bruijn reduction.
```

THEOREM: reduction-translates
$(\text{ntermp}(a) \wedge \text{nreduction}(a, b, \mathit{list}))$
$\rightarrow \quad \text{reduction}(\text{translate}(a, \textbf{nil}), \text{translate}(b, \textbf{nil}), \text{trans-list}(\mathit{list}))$

```
;Finds the largest free-variable in the de Bruijn term X.
```

DEFINITION:
find-m (x, n)
$=$ **if** lambdap (x) **then** find-m (body (x), $1 + n$)
elseif combp (x)
then if find-m (left (x), n) $<$ find-m (right (x), n)
then find-m (right (x), n)
else find-m (left (x), n) **endif**
else $x - n$ **endif**

```
;Translates the de Bruijn term X into a standard term given stack level N,
;and that M is the largest free variable in X.  Very tricky!
```

DEFINITION:
untrans (x, m, n)
$=$ **if** lambdap (x) **then** nlambda ($m + (1 + n)$, untrans (body (x), m, $1 + n$))
elseif combp (x)
then ncomb (untrans (left (x), m, n), untrans (right (x), m, n))
elseif $n < x$ **then** $x - (1 + n)$
elseif $x \not\simeq 0$ **then** $1 + (m + (n - x))$
else x **endif**

```
;X is a well-formed de Bruijn term.
```

DEFINITION:
termp (x)
$=$ **if** lambdap (x) **then** termp (body (x))
elseif combp (x) **then** termp (left (x)) $\wedge$ termp (right (x))
else $(x \not\simeq 0) \vee$ litatom (x) **endif**

THEOREM: not-free-in-untrans
$(\text{termp}(x) \land (\text{find-m}(x, n) \le m) \land ((m + n) < y))$
$\rightarrow \quad \text{not-free-in}(y, \text{untrans}(x, m, n))$

```
;An untranslated term is free for another untranslated term.
```

THEOREM: free-for-untrans
$(\text{termp}(x) \land \text{termp}(y) \land (n2 \le n1) \land (\text{find-m}(y, n2) \le m))$
$\rightarrow \quad \text{free-for}(\text{untrans}(x, m, n1), \text{untrans}(y, m, n2))$

DEFINITION:
trans-untrans-ind $(x, \mathit{bounds}, m, n)$
= **if** nlambdap (x)
then trans-untrans-ind (nbody (x), cons (nbind (x), bounds), m, $1 + n$)
elseif ncombp (x)
then trans-untrans-ind (nleft (x), bounds, m, n)
$\land$ trans-untrans-ind (nright (x), bounds, m, n)
else t endif

```
;Generates a list of bound variables of length N.
```

DEFINITION:
bindings (m, n)
= **if** $n \simeq 0$ **then nil**
else cons $(m + n$, bindings $(m, n - 1))$ **endif**

```
;A TRANSLATE followed by an UNTRANS leaves X unchanged (modulo bound variables).
```

THEOREM: trans-untrans
$((\text{length}(\mathit{bounds}) = n)$
$\land \quad \text{ntermp}(x)$
$\land \quad (\text{find-m}(\text{translate}(x, \mathit{bounds}), n) \le m))$
$\rightarrow \quad \text{alpha-equal}(x, \text{untrans}(\text{translate}(x, \mathit{bounds}), m, n), \mathit{bounds}, \text{bindings}(m, n))$

THEOREM: termp-ntermp
$\text{ntermp}(x) \rightarrow \text{termp}(\text{translate}(x, \mathit{bounds}))$

THEOREM: ntermp-termp
$\text{termp}(x) \rightarrow \text{ntermp}(\text{untrans}(x, m, n))$

THEOREM: index-freevar
$((n < x) \land ((x - n) \le m))$
$\rightarrow \quad (\text{index}((x - 1) - n, \text{bindings}(m, n)) = x)$

```
;An UNTRANS followed by a TRANSLATE doesn't change a thing.
```

THEOREM: untrans-trans
(termp (x) $\wedge$ (find-m $(x,\ n) \leq m$))
$\rightarrow$ (translate (untrans $(x,\ m,\ n)$, bindings $(m,\ n)$) $= x$)

THEOREM: untrans-subst
(termp (x)
$\wedge$ termp (y)
$\wedge$ lambdap (x)
$\wedge$ (find-m $(y,\ n) \leq m$)
$\wedge$ (find-m $(x,\ n) \leq m$))
$\rightarrow$ (translate (nsubst (nbody (untrans $(x,\ m,\ n)$),
untrans $(y,\ m,\ n)$,
nbind (untrans $(x,\ m,\ n)$)),
bindings $(m,\ n)$)
$=$ subst (body (x), y, `1`))

THEOREM: litatom-translate
litatom (translate $(a,\ \mathit{bounds})$) $\rightarrow$ (translate $(a,\ \mathit{bounds}) = a$)

```
;If A and B have the same translation, then they are alpha-equal.
```

THEOREM: translate-alpha
(ntermp (a) $\wedge$ ntermp (b) $\wedge$ (translate $(a,\ \mathit{bounds})$ = translate $(b,\ \mathit{bounds2})$))
$\rightarrow$ alpha-equal $(a,\ b,\ \mathit{bounds},\ \mathit{bounds2})$

THEOREM: termp-bump
termp (x) $\rightarrow$ termp (bump $(x,\ n)$)

THEOREM: termp-subst
(termp (x) $\wedge$ termp (y) $\wedge$ $(n \not\simeq \mathtt{0})$) $\rightarrow$ termp (subst $(x,\ y,\ n)$)

```
;Returns an instruction corresponding to a beta-step.
```

DEFINITION:
make-bstep $(a,\ b)$
$=$ **if** $a = b$ **then nil**
elseif lambdap (a) $\wedge$ lambdap (b) **then** make-bstep (body (a), body (b))
elseif combp (a)
then if lambdap (left (a))
$\wedge$ (b = subst (body (left (a)), right (a), `1`))
then list (**nil**, **nil**, `'reduce`)
else list (make-bstep (left (a), left (b)),
make-bstep (right (a), right (b))) **endif**
else nil endif

```
;Applies the instruction to a term.
```

Definition:
bstep $(w,\ x)$
$=$ **if** lambdap (x) **then** lambda (bstep $(w,$ body $(x)))$
elseif combp (x)
then if lambdap (left $(x)) \wedge$ (command $(w) =$ `'reduce`)
then subst (body (left (x)), right (x), `1`)
else comb (bstep (left-instrs (w), left (x)),
bstep (right-instrs (w), right (x))) **endif**
else x **endif**

```
;The instruction works.
```

Theorem: make-bstep-beta-step
beta-step $(a,\ b) \rightarrow$ (bstep (make-bstep $(a,\ b),\ a) = b$)

```
;A standard beta-step.
```

Definition:
nbstep $(w,\ x)$
$=$ **if** nlambdap (x) **then** nlambda (nbind (x), nbstep $(w,$ nbody $(x)))$
elseif ncombp (x)
then if nlambdap (nleft $(x)) \wedge$ (command $(w) =$ `'reduce`)
then nsubst (nbody (nleft (x)), nright (x), nbind (nleft (x)))
else ncomb (nbstep (left-instrs (w), nleft (x)),
nbstep (right-instrs (w), nright (x))) **endif**
else x **endif**

Theorem: lambdap-untrans
termp $(x) \rightarrow$ (nlambdap (untrans $(x,\ m,\ n)) =$ lambdap (x))

Definition:
find-m-subst-ind $(x,\ y,\ m,\ n,\ ind)$
$=$ **if** lambdap (x)
then find-m-subst-ind (body (x), bump $(y,$ `0`$),\ m,\ 1 + n,\ 1 + ind)$
elseif combp (x)
then find-m-subst-ind (left $(x),\ y,\ m,\ n,\ ind)$
$\wedge$ find-m-subst-ind (right $(x),\ y,\ m,\ n,\ ind)$
else t endif

Theorem: find-m-bump
find-m $(y,\ n) \not<$ find-m (bump $(y,\ count),\ 1 + n)$

Theorem: diff-zero
$(m \leq n) \rightarrow ((m - n) = \mathtt{0})$

DEFINITION:
untrans-bstep-ind $(a,\ b,\ m,\ n)$

$$
\begin{array}{ll}
= & \textbf{if } a = b \textbf{ then t} \\
& \textbf{elseif } \text{lambdap}(a) \land \text{lambdap}(b) \\
& \textbf{then } \text{untrans-bstep-ind}(\text{body}(a),\ \text{body}(b),\ m,\ 1 + n) \\
& \textbf{elseif } \text{combp}(a) \\
& \textbf{then if } \text{lambdap}(\text{left}(a)) \\
& \qquad\qquad \land \quad (b = \text{subst}(\text{body}(\text{left}(a)),\ \text{right}(a),\ \mathtt{1})) \textbf{ then t} \\
& \qquad\quad \textbf{else } \text{untrans-bstep-ind}(\text{left}(a),\ \text{left}(b),\ m,\ n) \\
& \qquad\qquad\quad \land \quad \text{untrans-bstep-ind}(\text{right}(a),\ \text{right}(b),\ m,\ n) \textbf{ endif} \\
& \textbf{else t endif}
\end{array}
$$

THEOREM: nbstep-nil
$(w \simeq \textbf{nil}) \rightarrow (\text{nbstep}(w,\ a) = a)$

```
;A beta-step translates back.
```

THEOREM: untrans-beta-step
$(\text{beta-step}(a,\ b) \land \text{termp}(a) \land (\text{find-m}(a,\ n) \leq m))$
$\rightarrow \quad \text{nbeta-step}(\text{untrans}(a,\ m,\ n),\ \text{nbstep}(\text{make-bstep}(a,\ b),\ \text{untrans}(a,\ m,\ n)))$

THEOREM: find-m-bump2
$(\mathit{count} \leq n) \rightarrow (\text{find-m}(\text{bump}(y,\ \mathit{count}),\ 1 + n) = \text{find-m}(y,\ n))$

THEOREM: termp-beta-step
$(\text{beta-step}(a,\ b) \land \text{termp}(a)) \rightarrow \text{termp}(b)$

THEOREM: termp-reduction
$(\text{reduction}(a,\ b,\ \mathit{list}) \land \text{termp}(a)) \rightarrow \text{termp}(b)$

THEOREM: find-m-subst
$((\text{find-m}(x,\ 1 + n) \leq m) \land (\text{find-m}(y,\ n) \leq m) \land (\mathit{ind} \leq n))$
$\rightarrow \quad (m \not< \text{find-m}(\text{subst}(x,\ y,\ 1 + \mathit{ind}),\ n))$

```
;What a beta-step translates back into is alpha-equal to what would've
;been gotten by doing the step directly in the standard form.
```

THEOREM: untrans-subst-nsubst

$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{termp}(x)} \\
\land & \text{termp}(y) \\
\land & \text{lambdap}(x) \\
\land & (\text{find-m}(y,\ n) \leq m) \\
\land & (\text{find-m}(x,\ n) \leq m)) \\
\rightarrow & \text{alpha-equal}(\text{nsubst}(\text{nbody}(\text{untrans}(x,\ m,\ n)), \\
& \qquad\qquad\qquad\qquad \text{untrans}(y,\ m,\ n), \\
& \qquad\qquad\qquad\qquad \text{nbind}(\text{untrans}(x,\ m,\ n))),
\end{array}
$$

untrans (subst (body (x), y, 1), m, n),
bindings (m, n),
bindings (m, n))

THEOREM: untrans-nbstep
(termp (x) $\land$ (find-m (x, n) $\leq$ m))
$\rightarrow$ alpha-equal (nbstep (w, untrans (x, m, n)),
untrans (bstep (w, x), m, n),
bindings (m, n),
bindings (m, n))

THEOREM: another-find-m-subst
((find-m (y, n) $\leq$ m) $\land$ (find-m (x, $1 + n$) $\leq$ m) $\land$ ($ind \leq n$))
$\rightarrow$ ($m \not<$ find-m (subst (x, y, $1 + ind$), n))

THEOREM: untrans-nbstep-zero
(beta-step (x, y) $\land$ termp (x) $\land$ (find-m (x, 0) $\leq$ m))
$\rightarrow$ alpha-equal (nbstep (make-bstep (x, y), untrans (x, m, 0)),
untrans (y, m, 0),
nil,
nil)

THEOREM: beta-step-find-m
(beta-step (a, b) $\land$ (find-m (a, n) $\leq$ m)) $\rightarrow$ ($m \not<$ find-m (b, n))

THEOREM: another-beta-step-find-m
beta-step (a, b) $\rightarrow$ (find-m (a, n) $\not<$ find-m (b, n))

THEOREM: reduction-find-m
reduction (a, b, *list*) $\rightarrow$ (find-m (a, n) $\not<$ find-m (b, n))

EVENT: Enable make-w; name this event 'g0350'.

EVENT: Enable make-reduction; name this event 'g0351'.

```
;Standard term corresponding to MAKE-W.
```

DEFINITION:
make-n-w (x, y, z, *list1*, *list2*)
= untrans (make-w (translate (x, **nil**),
translate (y, **nil**),
translate (z, **nil**),
trans-list (*list1*),
trans-list (*list2*)),
find-m (translate (x, **nil**), 0),
0)

THEOREM: alpha-equal-commutes
alpha-equal $(a,\ b,\ x,\ y)$ = alpha-equal $(b,\ a,\ y,\ x)$

DEFINITION:
end-cons $(x,\ y)$
= **if** listp (y) **then** cons (car (y), end-cons $(x$, cdr $(y)))$
else cons $(x$, **nil**$)$ **endif**

THEOREM: end-cons-nreduction
(nreduction $(a,\ b,\ \mathit{list})$ $\wedge$ nstep $(\mathit{a1},\ a)$)
$\rightarrow$ nreduction $(\mathit{a1},\ b$, end-cons $(a,\ \mathit{list}))$

```
;Translates reductions back.
```

DEFINITION:
make-nreduction $(a,\ b,\ w,\ m)$
= **if** listp (w)
then cons (nbstep (make-bstep (car (w), b), untrans (car (w), m, 0)),
cons (untrans (car (w), m, 0),
make-nreduction $(a$, car (w), cdr (w), $m)))$
else cons (untrans $(b,\ m,\ 0)$,
cons (nbstep (make-bstep $(a,\ b)$, untrans $(a,\ m,\ 0)$), **nil**)) **endif**

```
;Reductions translate.
```

THEOREM: reduction-make-nreduction
(reduction $(a,\ b,\ \mathit{list})$ $\wedge$ termp (a) $\wedge$ (find-m $(a,\ 0) \leq m$))
$\rightarrow$ nreduction (untrans $(a,\ m,\ 0)$, untrans $(b,\ m,\ 0)$, make-nreduction $(a,\ b,\ \mathit{list},\ m)$)

```
;Standard convergent reduction corresponding to MAKE-REDUCTION.
```

DEFINITION:
nmake-reduction $(x,\ y,\ z,\ \mathit{list1},\ \mathit{list2})$
= end-cons (untrans (translate $(y$, **nil**$)$, find-m (translate $(x$, **nil**$)$, 0), 0),
make-nreduction (translate $(y$, **nil**$)$,
make-w (translate $(x$, **nil**$)$,
translate $(z$, **nil**$)$,
translate $(y$, **nil**$)$,
trans-list $(\mathit{list2})$,
trans-list $(\mathit{list1})$),
make-reduction (translate $(x$, **nil**$)$,
translate $(y$, **nil**$)$,
translate $(z$, **nil**$)$,
trans-list $(\mathit{list1})$,
trans-list $(\mathit{list2})$),
find-m (translate $(x$, **nil**$)$, 0)))

THEOREM: trans-untrans-nil
(ntermp (x) $\wedge$ (find-m (translate $(x,$ **nil**$)$, `0`) $\leq m$))
$\rightarrow$ alpha-equal $(x,$ untrans (translate $(x,$ **nil**$)$, m, `0`), **nil**, **nil**)

```
;Church-Rosser theorem for standard representation.  FINALLY-CHURCH-ROSSER
;gives the complete form.
```

THEOREM: standard-church-rosser
(nreduction $(x,\ y,$ *list1*$)$ $\wedge$ nreduction $(x,\ z,$ *list2*$)$ $\wedge$ ntermp (x))
$\rightarrow$ nreduction $(y,$
make-n-w $(x,\ z,\ y,$ *list2*, *list1*$)$,
nmake-reduction $(x,\ y,\ z,$ *list1*, *list2*$)$)

THEOREM: both-make-n-w-are-same
(ntermp (x) $\wedge$ nreduction $(x,\ y,$ *list1*$)$ $\wedge$ nreduction $(x,\ z,$ *list2*$)$)
$\rightarrow$ (make-n-w $(x,\ y,\ z,$ *list1*, *list2*$)$ = make-n-w $(x,\ z,\ y,$ *list2*, *list1*$)$)

THEOREM: finally-church-rosser
(ntermp (x) $\wedge$ nreduction $(x,\ y,$ *list1*$)$ $\wedge$ nreduction $(x,\ z,$ *list2*$)$)
$\rightarrow$ (nreduction $(y,$
make-n-w $(x,\ z,\ y,$ *list2*, *list1*$)$,
nmake-reduction $(x,\ y,\ z,$ *list1*, *list2*$)$)
$\wedge$ nreduction $(z,$
make-n-w $(x,\ z,\ y,$ *list2*, *list1*$)$,
nmake-reduction $(x,\ z,\ y,$ *list2*, *list1*$)$))

```
;Now, the Diamond property of walks done directly in the standard notation.
```

THEOREM: free-for-nsubst
(free-for $(exp,\ x)$ $\wedge$ free-for $(y,\ x)$) $\rightarrow$ free-for (nsubst $(exp,\ y,\ z)$, x)

THEOREM: not-free-in-nsubst
(not-free-in $(x,\ exp)$ $\wedge$ not-free-in $(x,\ y)$)
$\rightarrow$ not-free-in $(x,$ nsubst $(exp,\ y,\ z))$

THEOREM: not-free-in-nsubst1
(not-free-in $(x,\ y)$ $\wedge$ $(x \in \mathbf{N})$) $\rightarrow$ not-free-in $(x,$ nsubst $(exp,\ y,\ x))$

DEFINITION:
nwalk $(w,\ x)$
= **if** nlambdap (x) **then** nlambda (nbind (x), nwalk $(w,$ nbody $(x)))$
elseif ncombp (x)
then if nlambdap (nleft (x)) $\wedge$ (command (w) = `'reduce`)
then nsubst (nbody (nwalk (left-instrs (w), nleft (x))),
nwalk (right-instrs (w), nright (x)),
nbind (nwalk (left-instrs (w), nleft (x))))

else ncomb (nwalk (left-instrs (w), nleft (x)),
nwalk (right-instrs (w), nright (x))) **endif**
else x **endif**

THEOREM: not-free-in-walk
not-free-in (x, *exp*) $\rightarrow$ not-free-in (x, nwalk (w, *exp*))

THEOREM: transitivity-of-alpha-equal
(alpha-equal (a, b, x, y) $\land$ alpha-equal (b, c, y, z)) $\rightarrow$ alpha-equal (a, c, x, z)

THEOREM: free-for-in-walk
free-for (*exp*, x) $\rightarrow$ free-for (nwalk (w, *exp*), x)

DEFINITION:
free-for-walk (w, x)
= **if** nlambdap (x) **then** free-for-walk (w, nbody (x))
elseif ncombp (x)
then if nlambdap (nleft (x)) $\land$ (command (w) = `'reduce`)
then free-for (nbody (nleft (x)), nright (x))
$\land$ free-for-walk (left-instrs (w), nleft (x))
$\land$ free-for-walk (right-instrs (w), nright (x))
else free-for-walk (left-instrs (w), nleft (x))
$\land$ free-for-walk (right-instrs (w), nright (x)) **endif**
else t endif

THEOREM: nlistp-free-for-walk
($w \simeq$ **nil**) $\rightarrow$ free-for-walk (w, x)

DEFINITION:
free-for-trans-ind (w, x, *bounds*)
= **if** nlambdap (x)
then free-for-trans-ind (w, nbody (x), cons (nbind (x), *bounds*))
elseif ncombp (x)
then free-for-trans-ind (left-instrs (w), nleft (x), *bounds*)
$\land$ free-for-trans-ind (right-instrs (w), nright (x), *bounds*)
else t endif

THEOREM: nbody-untrans
(ntermp (x) $\land$ lambdap (translate (x, *bounds*)))
$\rightarrow$ (nbody (untrans (translate (x, *bounds*), m, n))
= untrans (translate (nbody (x), cons (nbind (x), *bounds*)), m, $1 + n$))

THEOREM: free-for-walk-untrans-trans
(ntermp (x) $\land$ (find-m (translate (x, *bounds*), length (*bounds*)) $\leq m$))
$\rightarrow$ free-for-walk (w, untrans (translate (x, *bounds*), m, length (*bounds*)))

Theorem: free-for-in-walk1
$\text{free-for}\,(x,\ y) \rightarrow \text{free-for}\,(\text{nwalk}\,(\mathit{w1},\ x),\ \text{nwalk}\,(\mathit{w2},\ y))$

Theorem: alpha-equal-nsubst
$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{ntermp}\,(x)} \\
\ \wedge & \text{ntermp}\,(y) \\
\ \wedge & \text{ntermp}\,(\mathit{x1}) \\
\ \wedge & \text{ntermp}\,(\mathit{y1}) \\
\ \wedge & \text{nlambdap}\,(x) \\
\ \wedge & \text{nlambdap}\,(\mathit{x1}) \\
\ \wedge & \text{free-for}\,(\text{nbody}\,(x),\ y) \\
\ \wedge & \text{free-for}\,(\text{nbody}\,(\mathit{x1}),\ \mathit{y1}) \\
\ \wedge & \text{alpha-equal}\,(x,\ \mathit{x1},\ \mathit{bnd},\ \mathit{bnd1}) \\
\ \wedge & \text{alpha-equal}\,(y,\ \mathit{y1},\ \mathit{bnd},\ \mathit{bnd1})) \\
\rightarrow & \text{alpha-equal}\,(\text{nsubst}\,(\text{nbody}\,(x),\ y,\ \text{nbind}\,(x)), \\
 & \qquad\qquad\quad \text{nsubst}\,(\text{nbody}\,(\mathit{x1}),\ \mathit{y1},\ \text{nbind}\,(\mathit{x1})), \\
 & \qquad\qquad\quad \mathit{bnd}, \\
 & \qquad\qquad\quad \mathit{bnd1})
\end{array}
$$

Theorem: nwalk-ntermp
$\text{ntermp}\,(a) \rightarrow \text{ntermp}\,(\text{nwalk}\,(w,\ a))$

Theorem: nlambdap-nwalk
$\text{nlambdap}\,(a) \rightarrow \text{nlambdap}\,(\text{nwalk}\,(w,\ a))$

Theorem: free-for-nwalk2
$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{nlambdap}\,(x) \wedge \text{free-for}\,(\text{nbody}\,(x),\ y))} \\
\rightarrow & \text{free-for}\,(\text{nbody}\,(\text{nwalk}\,(\mathit{w1},\ x)),\ \text{nwalk}\,(\mathit{w2},\ y))
\end{array}
$$

Theorem: nwalk-alpha
$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{ntermp}\,(a)} \\
\ \wedge & \text{ntermp}\,(b) \\
\ \wedge & \text{alpha-equal}\,(a,\ b,\ x,\ y) \\
\ \wedge & \text{free-for-walk}\,(w,\ a) \\
\ \wedge & \text{free-for-walk}\,(w,\ b)) \\
\rightarrow & \text{alpha-equal}\,(\text{nwalk}\,(w,\ a),\ \text{nwalk}\,(w,\ b),\ x,\ y)
\end{array}
$$

Theorem: nsubst-not-free-in
$\text{not-free-in}\,(x,\ a) \rightarrow (\text{nsubst}\,(a,\ b,\ x) = a)$

Theorem: nsubst-nsubst
$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{free-for}\,(a,\ b) \wedge \text{not-free-in}\,(x,\ c) \wedge (x \neq y))} \\
\rightarrow & (\text{nsubst}\,(\text{nsubst}\,(a,\ b,\ x),\ c,\ y) = \text{nsubst}\,(\text{nsubst}\,(a,\ c,\ y),\ \text{nsubst}\,(b,\ c,\ y),\ x))
\end{array}
$$

Definition:

sub-nwalk (w, m, u, n)
$=$ **if** nlambdap (m)
then if nbind $(m) = n$ **then** w
else sub-nwalk $(w,$ nbody $(m), u, n)$ **endif**
elseif ncombp (m)
then if (command $(w) =$ `'reduce`) $\land$ nlambdap (nleft (m))
then list (sub-nwalk (left-instrs (w), nleft $(m), u, n)$,
sub-nwalk (right-instrs (w), nright $(m), u, n)$,
`'reduce`)
else list (sub-nwalk (left-instrs (w), nleft $(m), u, n)$,
sub-nwalk (right-instrs (w), nright $(m), u, n)$) **endif**
elseif $m \in \mathbf{N}$
then if $m = n$ **then** u
else w **endif**
else w **endif**

THEOREM: nlambdap-nsubst
nlambdap $(x) \rightarrow$ nlambdap (nsubst (x, y, z))

THEOREM: nsubst-twice
free-for $(a, b) \rightarrow$ (nsubst (nsubst $(a, b, x), c, x) =$ nsubst $(a,$ nsubst $(b, c, x), x)$)

THEOREM: nwalk-nsubst
(free-for $(m, x) \land$ free-for-walk (w, m))
$\rightarrow$ (nwalk (sub-nwalk (w, m, u, n), nsubst (m, x, n))
$=$ nsubst (nwalk (w, m), nwalk (u, x), n))

DEFINITION:
free-forx (a, b, x)
$=$ **if** not-free-in (x, a) **then t**
elseif nlambdap (a)
then if nbind $(a) = x$ **then t**
else not-free-in (nbind $(a), b) \land$ free-forx (nbody $(a), b, x)$ **endif**
elseif ncombp (a)
then free-forx (nleft $(a), b, x) \land$ free-forx (nright $(a), b, x$)
else t endif

THEOREM: nsubst-nsubstx
(free-forx $(a, b, x) \land$ not-free-in $(x, c) \land (x \neq y)$)
$\rightarrow$ (nsubst (nsubst $(a, b, x), c, y) =$ nsubst (nsubst (a, c, y), nsubst $(b, c, y), x$))

THEOREM: nsubst-twicex
free-forx (a, b, x)
$\rightarrow$ (nsubst (nsubst $(a, b, x), c, x) =$ nsubst $(a,$ nsubst $(b, c, x), x)$)

THEOREM: free-for-free-forx
free-for $(a, b) \rightarrow$ free-forx (a, b, x)

DEFINITION:
free-forx-walk (w, x)
$=$ **if** nlambdap (x) **then** free-forx-walk $(w,$ nbody $(x))$
elseif ncombp (x)
then if nlambdap (nleft $(x)) \wedge$ (command $(w) =$ `'reduce`)
then free-forx (nbody (nwalk (left-instrs (w), nleft $(x))$),
nwalk (right-instrs (w), nright (x)),
nbind (nwalk (left-instrs (w), nleft (x))))
$\wedge$ free-forx-walk (left-instrs (w), nleft (x))
$\wedge$ free-forx-walk (right-instrs (w), nright (x))
else free-forx-walk (left-instrs (w), nleft (x))
$\wedge$ free-forx-walk (right-instrs (w), nright (x)) **endif**
else t endif

THEOREM: free-for-free-forx-walk
free-for-walk $(w, x) \rightarrow$ free-forx-walk (w, x)

THEOREM: free-forx-nsubst
(free-forx $(x, m, n) \wedge$ free-forx $(x, y, z) \wedge$ not-free-in (z, m))
$\rightarrow$ free-forx (nsubst (x, m, n), y, z)

THEOREM: free-forx-nsubst2
(free-for $(x, m) \wedge$ free-forx $(x, y, z)) \rightarrow$ free-forx $(x$, nsubst (y, m, n), $z)$

THEOREM: nwalk-nlambda1
nlambdap $(x) \rightarrow$ (nbody (nwalk (u, x)) $=$ nwalk $(u$, nbody (x)))

THEOREM: nwalk-nlambda2
nlambdap $(x) \rightarrow$ (nbind (nwalk (u, x)) $=$ nbind (x))

THEOREM: nwalk-nsubstx
(free-for $(m, x) \wedge$ free-forx-walk (w, m))
$\rightarrow$ (nwalk (sub-nwalk (w, m, u, n), nsubst (m, x, n))
$=$ nsubst (nwalk (w, m), nwalk (u, x), n))

DEFINITION:
make-nwalk (m, u, v)
$=$ **if** nlambdap (m) **then** make-nwalk (nbody (m), u, v)
elseif ncombp (m)
then if (command $(u) =$ `'reduce`) $\wedge$ nlambdap (nleft (m))
then sub-nwalk (make-nwalk (nleft (m), left-instrs (u), left-instrs (v)),
nbody (nwalk (left-instrs (u), nleft (m))),

```
                          make-nwalk (nright (m),
                                      right-instrs (u),
                                      right-instrs (v)),
                          nbind (nwalk (left-instrs (u), nleft (m))))
        elseif (command (v) = 'reduce) ∧ nlambdap (nleft (m))
        then list (make-nwalk (nleft (m), left-instrs (u), left-instrs (v)),
                   make-nwalk (nright (m), right-instrs (u), right-instrs (v)),
                   'reduce)
        else list (make-nwalk (nleft (m),
                               left-instrs (u),
                               left-instrs (v)),
                  make-nwalk (nright (m),
                               right-instrs (u),
                               right-instrs (v))) endif
    else u endif
```

THEOREM: free-for-sub-nwalk
(free-for $(m,\ x)$ $\wedge$ free-forx-walk $(w,\ m)$ $\wedge$ free-forx-walk $(u,\ x)$)
$\rightarrow$ free-forx-walk (sub-nwalk $(w,\ m,\ u,\ n)$, nsubst $(m,\ x,\ n)$)

THEOREM: free-forx-make-nwalk
(free-for-walk $(u,\ m)$ $\wedge$ free-for-walk $(v,\ m)$)
$\rightarrow$ free-forx-walk (make-nwalk $(m,\ u,\ v)$, nwalk $(u,\ m)$)

`;The Diamond property of beta-walks.`

THEOREM: diamond-of-beta
(free-for-walk $(u,\ m)$ $\wedge$ free-for-walk $(v,\ m)$)
$\rightarrow$ (nwalk (make-nwalk $(m,\ u,\ v)$, nwalk $(u,\ m)$)
$=$ nwalk (make-nwalk $(m,\ v,\ u)$, nwalk $(v,\ m)$))

Index