```
#|

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

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

 NO WARRANTY

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

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

|#

; ------------------------------------------------------------
; was lr-eval5-1.events
; ------------------------------------------------------------
```

EVENT: Start with the library `"app-c-d-e"` using the compiled version.

THEOREM: axiom-53
subrp (*fn*) → (formals (*fn*) = **f**)

EVENT: Disable proper-p-statep-restructuring.

```
;; Function for testing s->r
```

DEFINITION:
change-elements (*list*)
= **if** listp (*list*)
  then if truep (car (*list*)) **then** cons (FALSE, change-elements (cdr (*list*)))
    else cons (TRUE, change-elements (cdr (*list*))) **endif**

elseif truep (*list*) **then** FALSE
else TRUE **endif**

EVENT: Disable deposit.

EVENT: Disable fetch.

EVENT: Disable add-addr.

EVENT: Disable sub-addr.

EVENT: Disable offset.

EVENT: Disable area-name.

EVENT: Disable errorp.

EVENT: Disable p-current-program.

```
;; The following is inspired by the lemma length-put of Piton.
;; Now in Piton-basis  A. Flatau  8-Oct-1990
;(prove-lemma MY-LENGTH-PUT (rewrite)
;  (equal (length (put val n lst))
;  (if (lessp n (length lst))
;      (length lst)
;      (add1 n)))
;  ((enable put)))
;
;(disable my-length-put)

;; This is similar to the lemma GET-PUT from Piton, but for the commented
;; out hypothesis.
```

THEOREM: my-get-put
$((k \in \mathbf{N}) \wedge (n \in \mathbf{N}))$
$\rightarrow$ (get (*k*, put (*val*, *n*, *lst*))
= **if** $k = n$ **then** *val*
else get (*k*, *lst*) **endif**)

EVENT: Disable my-get-put.

THEOREM: listp-cdr-p-frame
listp (cdr (p-frame (*bindings*, *ret-pc*)))

THEOREM: equal-cddr-p-frame-nil
cddr (p-frame (*bindings*, *ret-pc*)) = **nil**

```
#||
;; The following is used to test handling of temp variables
(defn FOO (state name)
  (let ((prog (app name state)))
    (cons state (cons (car prog) (cons (cadr prog) (caddr prog))))))

;(setq ss
;      (logic->s '(change-elements (cons '*1*true (app x y)))
;  '((x . (*1*true *1*true . *1*false))
;    (y . (*1*true . *1*false)))
;   '(change-elements app)))
;(setq lrs (s->lr ss 'main 50 50 50 32))
;(setq foop
;      '(FOO (STATE NAME)
;     ((APP NAME STATE)
;      (CDR ((TEMP-FETCH) (APP NAME STATE))))
;     (CONS STATE
;   (CONS (CAR ((TEMP-EVAL) (APP NAME STATE)))
; (CONS (CAR ((TEMP-EVAL)
;     (CDR ((TEMP-FETCH) (APP NAME STATE)))))
;       (CAR (CDR ((TEMP-FETCH)
;  (CDR ((temp-fetch)
;        (APP NAME STATE)))))))))))
;
;(setq ss1 (s-state (s-expr ss)
;    (s-params ss)
;    (s-temps ss)
;    (s-consts ss)
;    (put-assoc (cdr foop) 'foo (s-progs ss))
;    'run))
;
;(setq ss2 (s-state '(FOO (CHANGE-ELEMENTS (CONS '(ADDR (heap . 4))
; (APP ((temp-eval) X) Y)))
;  ((temp-fetch) X))
;    (s-params ss1)
;    (make-temps-entries '(x))
;    (s-consts ss1)
;    (s-progs ss1)
```

```
;     'run))
||#
```

DEFINITION:
s-l-eval-equiv-hyps (*flag*, *s*, *c*)
= (s-good-statep (*s*, *c*)
  ∧ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
  ∧ (s-err-flag (s-eval (*flag*, *s*, *c*)) = 'run))

DEFINITION:
s-l-eval-flag-run-hyps (*flag*, *s*, *c*)
= (s-good-statep (*s*, *c*)
  ∧ s-all-temps-setp (*flag*,
                      if *flag* = 'list **then** s-expr-list (*s*)
                      else s-expr (*s*) **endif**,
                      temp-alist-to-set (s-temps (*s*)))
  ∧ s-all-progs-temps-setp (s-progs (*s*))
  ∧ **if** *flag* = 'list
    then f ∉ l-eval (*flag*,
                         s-expand-temps (*flag*, s-expr-list (*s*)),
                         s-params (*s*),
                         c)
    else l-eval (*flag*,
                     s-expand-temps (*flag*, s-expr (*s*)),
                     s-params (*s*),
                     c) **endif**
  ∧ s-check-temps-setp (s-temps (*s*)))

```
;; ***** The LR-level (R for Resource, L for logic).  *****
;; We used to have an LR-STATE shell.  Now we just use a P-STATE shell.
;; However we refer to LR-STATES which are P-STATEs with LR level programs.
;; The function LR->P compiles an LR-STATE to a Piton state, by compiling
;; the programs and converting the P-PC to a Piton PC.
;; We use P-STATE shells instead of LR-STATE shell because we used to have
;; define functions analogous to P-OBJECTP (and functions that called
;; P-OBJECTP) that took LR-STATES or parts thereof.

;; We use the Piton notion of a PROPER state.  It should be the case that
;; all the LR-STATEs we are interested in are PROPER-P-STATEPs after we
;; apply LR->P to them.

;; An LR PC object is a combination of a Piton PC object and an S level
;; S-PNAME and S-POS.  The translation of (s-pname s) and (s-pos s) from
;; the S level is: (TAG 'PC (CONS (S-PNAME S) (S-POS S)))
```

```
;; Each element of P-PROG-SEGMENT is a program.  A program is a list
;; of the form:
;;
;; (name (formal1 formal2 ... formaln)
;;       ((temp1 init1)
;;        ...
;;        (tempk initk))
;;       body)
;;
;; The name and each formal and temp is a symbol.  The initial values
;; of the temps are tagged values.  Body is a form similar to that for
;; the S level, but temporary expressions have been replaced the name of
;; a temporary variable added to them
;; e.g. ((S-TEMP-EVAL) <expr>) -> ((S-TEMP-EVAL) <expr> <var>).
;; In the case of (S-TEMP-FETCH) <expr> is never used but we put it
;; in for consistency and so it is easier to convert back to s-states.
;; Also the numbers in the S level quote constructs have been replaced
;; by data-addresses that should contain pointers to the appropriate
;; structure in the heap.

;; Roughly speaking, a function application of FUN binds the formals to
;; the top n elements of the temp-stk (removing them from that stack and
;; building a ctrl-stk frame), binds the temps to the corresponding tagged
;; values (also in the ctrl-stk frame), and executes each instruction.

;; Producing LR-code from S-code.
```

DEFINITION: LR-UNDEFINED-TAG = 0

```
; Used in node to indicate
; uninitialized temporary variable
```

DEFINITION: LR-INIT-TAG = 1

```
; Used in initial nodes that have
; not been used
```

DEFINITION: LR-FALSE-TAG = 2

DEFINITION: LR-TRUE-TAG = 3

DEFINITION: LR-ADD1-TAG = 4

DEFINITION: LR-CONS-TAG = 5

DEFINITION: LR-PACK-TAG = 6

DEFINITION: LR-MINUS-TAG = 7

DEFINITION: LR-HEAP-NAME = 'heap

DEFINITION: LR-NODE-SIZE = 4

DEFINITION: LR-UNDEF-ADDR = tag ('addr, '(heap . 0))

DEFINITION: LR-F-ADDR = add-addr (LR-UNDEF-ADDR, LR-NODE-SIZE)

DEFINITION: LR-T-ADDR = add-addr (LR-F-ADDR, LR-NODE-SIZE)

DEFINITION: LR-0-ADDR = add-addr (LR-T-ADDR, LR-NODE-SIZE)

DEFINITION: LR-FP-ADDR = tag ('addr, '(free-ptr . 0))

DEFINITION: LR-ANSWER-ADDR = tag ('addr, '(answer . 0))

DEFINITION: lr-fetch-fp (*data-seg*) = fetch (LR-FP-ADDR, *data-seg*)

DEFINITION:
LR-MINIMUM-HEAP-SIZE = offset (add-addr (LR-0-ADDR, LR-NODE-SIZE))

```
;; The heap is a (presumably large) Piton data area.  It contains Nodes
;; which are four words.  One word is for the tag, one for the reference
;; count, and two for the contents.  Some data-types only require one word
;; for the contents (e.g. NUMBERPs) in that case one word is wasted.  Some
;; (user-defined) data-types require more than two words. In this case the
;; second word is a pointer to another node.  This contains up to three
;; words of data, the fourth word (if the data type needs more that four
;; words) is used to link another node with the same format.  The heap is
;; the Piton data area named HEAP.

;; LR-NEW-NODE returns another node to be stuck in memory
```

DEFINITION:
lr-new-node (*tag*, *ref-count*, *value1*, *value2*)
= list (*tag*, *ref-count*, *value1*, *value2*)

DEFINITION: LR-REF-COUNT-OFFSET = 1

DEFINITION: LR-CAR-OFFSET = 2

DEFINITION: LR-CDR-OFFSET = 3

DEFINITION: LR-UNPACK-OFFSET = 2

DEFINITION: LR-UNBOX-NAT-OFFSET = 2

DEFINITION: LR-NEGATIVE-GUTS-OFFSET = 2

DEFINITION:
lr-boundary-offsetp (*offset*) = ((*offset* **mod** LR-NODE-SIZE) = 0)

DEFINITION:
lr-boundary-nodep (*node*) = lr-boundary-offsetp (offset (*node*))

DEFINITION:
lr-nodep (*addr*, *data-seg*)
= ((type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ adpp (untag (*addr*), *data-seg*)
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = LR-HEAP-NAME))

```
;; LR-GOOD-POINTERP checks that an addr is a node and its ref count field
;; is a natural.
```

DEFINITION:
lr-good-pointerp (*addr*, *data-seg*)
= (lr-nodep (*addr*, *data-seg*)
∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*))
= 'nat))

DEFINITION:
lr-expr (*p*) = cur-expr (offset (p-pc (*p*)), program-body (p-current-program (*p*)))

EVENT: Disable lr-expr.

DEFINITION:
lr-expr-list (*p*)
= restn (car (last (offset (p-pc (*p*)))),
cur-expr (butlast (offset (p-pc (*p*))),
program-body (p-current-program (*p*))))

EVENT: Disable lr-expr-list.

```
;;; Debugging Stuff.
```

```
DEFINITION:
mark-instr (instruction-list, n)
=  if n ≃ 0
   then cons (list ('pc->, car (instruction-list)), cdr (instruction-list))
   else cons (car (instruction-list),
              mark-instr (cdr (instruction-list), n − 1)) endif

DEFINITION:
fix-program-segment (programs, pc)
=  if listp (programs)
   then let prog be car (programs)
        in
        if car (prog) = area-name (pc)
        then cons (append (list (car (prog), cadr (prog), caddr (prog)),
                                mark-instr (program-body (prog), offset (pc))),
                  fix-program-segment (cdr (programs), pc))
        else cons (car (prog),
                   fix-program-segment (cdr (programs), pc)) endif endlet
   else nil endif

DEFINITION:
fix-data-segment (data-segment)
=  put-value (append (firstn (offset (lr-fetch-fp (data-segment)),
                              value (LR-HEAP-NAME, data-segment)),
                     length (value (LR-HEAP-NAME, data-segment))
                     −  offset (lr-fetch-fp (data-segment))),
              LR-HEAP-NAME,
              data-segment)

DEFINITION:
find-non-proper-instr (lst, name, p)
=  if listp (lst)
   then if legal-labelp (car (lst))
           ∧  proper-p-instructionp (unlabel (car (lst)), name, p)
        then find-non-proper-instr (cdr (lst), name, p)
        else car (lst) endif
   else nil endif

DEFINITION:
find-non-proper-programs (progs, p)
=  if listp (progs)
   then if proper-p-programp (car (progs), p)
        then cons (name (car (progs)),
                   find-non-proper-programs (cdr (progs), p))
        else cons (list ('not,
```

```
                    name (car (progs)),
                    find-non-proper-instr (program-body (car (progs)),
                                           name (car (progs)),
                                           p)),
              find-non-proper-programs (cdr (progs), p)) endif
    else nil endif
```

DEFINITION:

```
pps (state)
=   list ('p-state,
        p-pc (state),
        p-ctrl-stk (state),
        p-temp-stk (state),
        let p be p-current-program (state)
        in
        append (list (name (p), formal-vars (p), temp-var-dcls (p)),
                mark-instr (program-body (p), offset (p-pc (state)))) endlet,
        fix-data-segment (p-data-segment (state)),
        p-psw (state))
```

DEFINITION:

```
lr-nodify-tag (tag)
=   if untag (tag) = LR-FALSE-TAG then 'false
    elseif untag (tag) = LR-TRUE-TAG then 'true
    elseif untag (tag) = LR-ADD1-TAG then 'add1
    elseif untag (tag) = LR-CONS-TAG then 'cons
    elseif untag (tag) = LR-PACK-TAG then 'pack
    else 'unknown endif
```

DEFINITION:

```
lr-nodify (number, nodes, final)
=   if listp (nodes)
    then cons (list ('node,
                     number,
                     lr-nodify-tag (car (nodes)),
                     caddr (nodes),
                     cadddr (nodes)),
               lr-nodify (number + LR-NODE-SIZE, cddddr (nodes), final))
    else final endif
```

DEFINITION:

```
lr-fix-data-segment (data-seg)
=   put-value (lr-nodify (0,
                          firstn (offset (lr-fetch-fp (data-seg)),
                                  value (LR-HEAP-NAME, data-seg)),
```

```
                    length (value (LR-HEAP-NAME, data-seg))
                    -  offset (lr-fetch-fp (data-seg))),
                LR-HEAP-NAME,
                data-seg)

DEFINITION:
lrps (state)
=  p-state (p-pc (state),
              p-ctrl-stk (state),
              p-temp-stk (state),
              p-prog-segment (state),
              lr-fix-data-segment (p-data-segment (state)),
              p-max-ctrl-stk-size (state),
              p-max-temp-stk-size (state),
              p-word-size (state),
              p-psw (state))

;; Returns the object denoted by addr in the heap.

DEFINITION:
lr-abs (addr, data-seg, n)
=  if n ≃ 0 then nil
   else let tag be untag (fetch (addr, data-seg))
        in
        if tag = LR-FALSE-TAG then f
        elseif tag = LR-TRUE-TAG then t
        elseif tag = LR-ADD1-TAG
        then untag (fetch (add-addr (addr, LR-UNBOX-NAT-OFFSET),
                           data-seg))
        elseif tag = LR-CONS-TAG
        then cons (lr-abs (fetch (add-addr (addr, LR-CAR-OFFSET),
                                  data-seg),
                           data-seg,
                           n - 1),
                   lr-abs (fetch (add-addr (addr, LR-CDR-OFFSET),
                                  data-seg),
                           data-seg,
                           n - 1))
        elseif tag = LR-PACK-TAG
        then pack (lr-abs (fetch (add-addr (addr,
                                            LR-UNPACK-OFFSET),
                                  data-seg),
                           data-seg,
                           n - 1))
        else - untag (fetch (add-addr (addr,
```

LR-NEGATIVE-GUTS-OFFSET),
data-seg)) **endif endlet endif**

DEFINITION:
top-stk (*p-or-p-state*)
= **let** *temp-stk* **be if** p-statep (*p-or-p-state*)
then p-temp-stk (*p-or-p-state*)
else p-temp-stk (*p-or-p-state*) **endif**,
data-segment **be if** p-statep (*p-or-p-state*)
then p-data-segment (*p-or-p-state*)
else p-data-segment (*p-or-p-state*) **endif**
in
lr-abs (top (*temp-stk*), *data-segment*, 1000) **endlet**

```
;; This is accessed by the Piton accessors: NAME, FORMAL-VARS, TEMP-VAR-DCLS
;; and PROGRAM-BODY.  Also LOCAL-VARS.
```

DEFINITION:
lr-make-program (*name*, *formals*, *temps*, *body*)
= cons (*name*, cons (*formals*, cons (*temps*, *body*)))

```
#||
stolen from matt kaufmann's code for gensym, but modified to probably be
less useful but simplier.

here is a sequence of events for generating a new symbol.
the main function is near the end, and is called gensym.
gensym returns a pair the new symbol and the next number list to try.
here are some examples:

>(r-loop)

trace mode: off   abbreviated output mode:  on
type ? for help.
*(gensym (unpack 'a*) '(49) '(a*0 a*1 a*2 a*3))
 '(a*4 53)
*(gensym (unpack 'a*) '(53) '(a*0 a*1 a*2 a*3 a*4))
 '(a*5 54)
*(gensym (unpack 'a*) '(50) '(a*2))
 '(a*3 52)
*(gensym (unpack 'a*) '(50) '(a*0))
 '(a*2 51)
*(gensym (unpack 'a) '(48) '(a*0 a*1))
 '(a0 49)
*(gensym (unpack 'a) '(48) '(a b))
```

```
 '(a0 49)
*(gensym (unpack 'a*2*) '(51) '(a*2*3))
 '(a*2*4 53)
*(gensym (unpack 'b*) '(50) '(a*0 a*1 a*2 a*3))
 '(b*2 51)
*ok
exiting r-loop.
nil

||#
```

DEFINITION: ASCII-0 = 48

DEFINITION: ASCII-1 = 49

DEFINITION: ASCII-9 = 57

DEFINITION: ASCII-DASH = 45

DEFINITION: LIST-ASCII-0 = list (ASCII-0)

DEFINITION: LIST-ASCII-1 = list (ASCII-1)

DEFINITION:
increment-numlist (*numlist*)
= **if** listp (*numlist*)
then if car (*numlist*) = ASCII-9
then cons (ASCII-0, increment-numlist (cdr (*numlist*)))
else cons (1 + car (*numlist*), cdr (*numlist*)) **endif**
else LIST-ASCII-1 **endif**

DEFINITION:
make-symbol (*initial*, *digit-list*)
= pack (append (append (*initial*, *digit-list*), 0))

EVENT: Disable make-symbol.

DEFINITION:
count-codelist1 (*numlist*)
= **if** listp (*numlist*)
then car (*numlist*) + (10 ∗ count-codelist1 (cdr (*numlist*)))
else 0 **endif**

DEFINITION:
subseqp (*list1*, *list2*)
= ((length (*list2*) $\not<$ length (*list1*))
$\wedge$ (firstn (length (*list1*), *list2*) = *list1*))

EVENT: Disable subseqp.

DEFINITION:
count-codelist (*initial*, *ascii-list*)
= **if** subseqp (*initial*, *ascii-list*)
then count-codelist1 (restn (length (*initial*), *ascii-list*))
else 0 **endif**

EVENT: Disable count-codelist.

DEFINITION:
max-count-codelist (*initial*, *list*)
= **if** listp (*list*)
then max (count-codelist (*initial*, unpack (car (*list*))),
max-count-codelist (*initial*, cdr (*list*)))
else 0 **endif**

THEOREM: increment-num-list-count-code-list1
count-codelist1 (*num-list*) < count-codelist1 (increment-numlist (*num-list*))

THEOREM: subseqp-append
subseqp (plist (*x*), append (*x*, *anything*))

THEOREM: count-codelist-make-symbol
(*x* = make-symbol (*initial*, *num-list*))
$\rightarrow$ (count-codelist (plist (*initial*), unpack (*x*))
= count-codelist1 (*num-list*))

THEOREM: member-make-symbol-max-count-code-list
(make-symbol (*initial*, *num-list*) $\in$ *atom-list*)
$\rightarrow$ (max-count-codelist (plist (*initial*), *atom-list*)
$\not<$ count-codelist1 (*num-list*))

```
;; Returns a pair, the new symbol and the next number to use
```

DEFINITION:
gensym (*initial*, *num-list*, *atom-list*)
= **if** make-symbol (*initial*, *num-list*) $\in$ *atom-list*
then gensym (*initial*, increment-numlist (*num-list*), *atom-list*)
else cons (make-symbol (*initial*, *num-list*),
increment-numlist (*num-list*)) **endif**

THEOREM: gensym-is-new
car (gensym (*initial*, *num-list*, *atom-list*)) $\notin$ *atom-list*

```
;; MAKE-TEMP-NAME-ALIST takes a temps-alist triple a la S-TEMPS and
;; returns an alist with entries of the form:
;; (<temp expression> . <variable>) where <variable> is guaranteed to
;; occur only once in the resulting alist and is guaranteed not to occur
;; in FORMALS.
```

DEFINITION:
lr-make-temp-name-alist-1 (*initial*, *num-list*, *temp-list*, *formals*)
= **if** listp (*temp-list*)
then let *gensym* **be** gensym (*initial*, *num-list*, *formals*)
in
cons (cons (car (*temp-list*), car (*gensym*)),
lr-make-temp-name-alist-1 (*initial*,
cdr (*gensym*),
cdr (*temp-list*),
formals)) **endlet**
else nil endif

DEFINITION:
lr-make-temp-name-alist (*temp-list*, *formals*)
= lr-make-temp-name-alist-1 (unpack (`'t*`), LIST-ASCII-0, *temp-list*, *formals*)

DEFINITION:
lr-new-cons (*car*, *cdr*)
= lr-new-node (tag (`'nat`, LR-CONS-TAG), tag (`'nat`, `1`), *car*, *cdr*)

```
;; Deposit LIST of objects at ADDR, ADDR+1, ADDR+2, ... in DATA-SEG.
```

DEFINITION:
deposit-a-list (*list*, *addr*, *data-seg*)
= **if** listp (*list*)
then deposit (car (*list*),
addr,
deposit-a-list (cdr (*list*), add1-addr (*addr*), *data-seg*))
else *data-seg* **endif**

DEFINITION:
lr-init-heap-contents (*addr*, *size*)
= **if** *size* $\simeq$ `0` **then** list (tag (`'nat`, LR-INIT-TAG))
else append (lr-new-node (tag (`'nat`, LR-INIT-TAG),
add-addr (*addr*, LR-NODE-SIZE),
tag (`'nat`, `0`),

```
                                    tag ('nat, 0)),
                        lr-init-heap-contents (add-addr (addr, LR-NODE-SIZE),
                                                size − 1)) endif

DEFINITION:
lr-add-to-data-seg (data-seg, new-node)
= if (length (value (LR-HEAP-NAME, data-seg)) − 1)
     ≮  (offset (fetch (LR-FP-ADDR, data-seg)) + length (new-node))
  then deposit (fetch (add-addr (fetch (LR-FP-ADDR, data-seg),
                                        LR-REF-COUNT-OFFSET),
                                data-seg),
                       LR-FP-ADDR,
                       deposit-a-list (new-node,
                                       fetch (LR-FP-ADDR, data-seg),
                                       data-seg))
  else data-seg endif

DEFINITION:
lr-init-data-seg (heap-size)
= deposit-a-list (list (tag ('nat, LR-FALSE-TAG),
                        tag ('nat, 1),
                        LR-UNDEF-ADDR,
                        LR-UNDEF-ADDR),
                  LR-F-ADDR,
                  deposit-a-list (list (tag ('nat, LR-UNDEFINED-TAG),
                                        tag ('nat, 1),
                                        LR-UNDEF-ADDR,
                                        LR-UNDEF-ADDR),
                                  LR-UNDEF-ADDR,
                                  list (list (area-name (LR-FP-ADDR),
                                              add-addr (LR-F-ADDR,
                                                        LR-NODE-SIZE)),
                                        list (area-name (LR-ANSWER-ADDR),
                                              tag ('nat, 0)),
                                        cons (LR-HEAP-NAME,
                                              lr-init-heap-contents (tag ('addr,
                                                                          cons (LR-HEAP-NAME,
                                                                                0)),
                                                                     heap-size)))))

DEFINITION:
count-list (flag, object)
= if flag = 'list
  then if listp (object)
       then count-list (t, car (object))
```

```
                + count-list ('list, cdr (object))
        else 1 endif
    elseif listp (object)
    then 1 + (1 + (count-list (t, car (object))
                  + count-list (t, cdr (object))))
    elseif object ∈ N then 1 + count (object)
    else 1 endif
```

THEOREM: not-equal-0-count-list
count-list (*flag*, *object*) ≠ 0

THEOREM: lessp-count-list-cdr-count-list-whole
listp (*object*)
→ (count-list ('list, cdr (*object*)) < count-list ('list, *object*))

THEOREM: lessp-count-not-list-car-count-list-whole
listp (*object*)
→ (count-list (t, car (*object*)) < count-list ('list, *object*))

```
;; LR-COMPILE-QUOTE returns a pair, the new HEAP and the new TABLE.
```

DEFINITION:

```
lr-compile-quote (flag, object, heap, table)
=  if flag = 'list
   then if listp (object)
        then let car-pair be lr-compile-quote (t,
                                               car (object),
                                               heap,
                                               table)
             in
             lr-compile-quote ('list,
                               cdr (object),
                               car (car-pair),
                               cdr (car-pair)) endlet
        else cons (heap, table) endif
   elseif definedp (object, table) then cons (heap, table)
   elseif listp (object)
   then let pair be lr-compile-quote ('list,
                                      list (car (object), cdr (object)),
                                      heap,
                                      table)
        in
        cons (lr-add-to-data-seg (car (pair),
                                  lr-new-cons (cdr (assoc (car (object),
                                                           cdr (pair))),
```

```
                                                cdr (assoc (cdr (object),
                                                           cdr (pair)))),
                cons (cons (object, fetch (LR-FP-ADDR, car (pair))),
                           cdr (pair))) endlet
    elseif object ∈ N
    then cons (lr-add-to-data-seg (heap,
                                   lr-new-node (tag ('nat, LR-ADD1-TAG),
                                                tag ('nat, 1),
                                                tag ('nat, object),
                                                LR-UNDEF-ADDR)),
               cons (cons (object, fetch (LR-FP-ADDR, heap)), table))
    elseif truep (object)
    then cons (lr-add-to-data-seg (heap,
                                   lr-new-node (tag ('nat, LR-TRUE-TAG),
                                                tag ('nat, 1),
                                                LR-UNDEF-ADDR,
                                                LR-UNDEF-ADDR)),
               cons (cons (object, fetch (LR-FP-ADDR, heap)), table))
    else cons (heap, cons (cons (object, LR-UNDEF-ADDR), table)) endif

;; LR-DATA-SEG-TABLE-BODY returns a pair, the CAR is the extension of
;; DATA-SEG with any constants laid down in it, the CDR is an alist
;; mapping objects in the logic to addresses in the new DATA-SEG
;; where they are represented.  The initial TABLE is such an alist

DEFINITION:
lr-data-seg-table-body (flag, expr, data-seg, table)
=   if flag = 'list
    then if listp (expr)
         then let dst1 be lr-data-seg-table-body (t,
                                                  car (expr),
                                                  data-seg,
                                                  table)
              in
              lr-data-seg-table-body ('list,
                                      cdr (expr),
                                      car (dst1),
                                      cdr (dst1)) endlet
         else cons (data-seg, table) endif
    elseif listp (expr)
    then if (car (expr) = S-TEMP-FETCH)
          ∨  (car (expr) = S-TEMP-EVAL)
          ∨  (car (expr) = S-TEMP-TEST)
         then lr-data-seg-table-body (t, cadr (expr), data-seg, table)
```

```
        elseif car (expr) = 'quote
        then lr-compile-quote (t, cadr (expr), data-seg, table)
        else lr-data-seg-table-body ('list,
                                     cdr (expr),
                                     data-seg,
                                     table) endif
    else cons (data-seg, table) endif
```

DEFINITION:

```
lr-data-seg-table-list (progs, data-seg, table)
=   if listp (progs)
    then lr-data-seg-table-list (cdr (progs),
                                 car (lr-data-seg-table-body (t,
                                                              s-body (car (progs)),
                                                              data-seg,
                                                              table)),
                                 cdr (lr-data-seg-table-body (t,
                                                              s-body (car (progs)),
                                                              data-seg,
                                                              table)))
    else cons (data-seg, table) endif
```

DEFINITION:

```
lr-init-data-seg-table (params, data-seg, table)
=   if listp (params)
    then let ds-tab be lr-compile-quote (t, cdar (params), data-seg, table)
         in
         lr-init-data-seg-table (cdr (params),
                                 car (ds-tab),
                                 cdr (ds-tab)) endlet
    else cons (data-seg, table) endif
```

DEFINITION:

```
lr-data-seg-table (progs, params, heap-size)
=   let init-ds-table1 be lr-compile-quote ('list,
                                            list (t, 0),
                                            lr-init-data-seg (heap-size),
                                            list (cons (f, LR-F-ADDR)))
    in
    let init-ds-table2 be lr-init-data-seg-table (params,
                                                  car (init-ds-table1),
                                                  cdr (init-ds-table1))
    in
    lr-data-seg-table-list (progs,
                            car (init-ds-table2),
```

cdr (*init-ds-table2*)) **endlet endlet**

DEFINITION:
pair-formals-with-addresses (*formals*, *table*)
= **if** listp (*formals*)
then cons (cons (caar (*formals*), cdr (assoc (cdar (*formals*), *table*))),
pair-formals-with-addresses (cdr (*formals*), *table*))
else nil endif

DEFINITION:
lr-make-initial-temps (*temp-vars*)
= **if** listp (*temp-vars*)
then cons (cons (car (*temp-vars*), LR-UNDEF-ADDR),
lr-make-initial-temps (cdr (*temp-vars*)))
else nil endif

DEFINITION:
lr-initial-cstk (*params*, *temp-alist*, *table*, *pc*)
= list (p-frame (append (pair-formals-with-addresses (*params*, *table*),
lr-make-initial-temps (strip-cdrs (*temp-alist*))),
pc))

DEFINITION:
lr-compile-body (*flag*, *body*, *temp-alist*, *const-table*)
= **if** *flag* = `'list`
then if listp (*body*)
then cons (lr-compile-body (**t**, car (*body*), *temp-alist*, *const-table*),
lr-compile-body (`'list`,
cdr (*body*),
temp-alist,
const-table))
else nil endif
elseif listp (*body*)
then if (car (*body*) = S-TEMP-FETCH)
∨ (car (*body*) = S-TEMP-EVAL)
∨ (car (*body*) = S-TEMP-TEST)
then list (car (*body*),
lr-compile-body (**t**, cadr (*body*), *temp-alist*, *const-table*),
value (cadr (*body*), *temp-alist*))
elseif car (*body*) = `'quote`
then list (`'quote`, value (cadr (*body*), *const-table*))
else cons (car (*body*),
lr-compile-body (`'list`,
cdr (*body*),
temp-alist,

```
                                        const-table)) endif
    else body endif

DEFINITION:
lr-make-temp-var-dcls (temp-alist)
=   if listp (temp-alist)
    then cons (list (cdar (temp-alist), LR-UNDEF-ADDR),
               lr-make-temp-var-dcls (cdr (temp-alist)))
    else nil endif

DEFINITION:
lr-compile-programs (programs, const-table)
=   if listp (programs)
    then let prog be car (programs)
         in
         let temp-alist be lr-make-temp-name-alist (s-temp-list (prog),
                                                    s-formals (prog))
         in
         cons (lr-make-program (car (prog),
                                s-formals (prog),
                                lr-make-temp-var-dcls (temp-alist),
                                lr-compile-body (t,
                                                 s-body (prog),
                                                 temp-alist,
                                                 const-table)),
               lr-compile-programs (cdr (programs), const-table)) endlet endlet
    else nil endif

DEFINITION:
lr-p-c-size (flag, expr)
=   if flag = 'list
    then if listp (expr)
         then lr-p-c-size (t, car (expr))
              +  lr-p-c-size ('list, cdr (expr))
         else 0 endif
    elseif listp (expr)
    then if car (expr) = 'if
         then lr-p-c-size (t, cadr (expr))
              +  lr-p-c-size (t, caddr (expr))
              +  lr-p-c-size (t, cadddr (expr))
              +  4
         elseif car (expr) = S-TEMP-FETCH then 1
         elseif car (expr) = S-TEMP-EVAL
         then lr-p-c-size (t, cadr (expr)) + 1
         elseif car (expr) = S-TEMP-TEST
```

then lr-p-c-size (**t**, cadr (*expr*)) + 7
elseif car (*expr*) = 'quote **then** 1
else lr-p-c-size ('list, cdr (*expr*)) + 1 **endif**
else 1 **endif**

DEFINITION:
lr-p-c-size-list (*n*, *expr-list*)
= **if** *n* ≃ 0 **then** 0
elseif *n* < length (*expr-list*)
then lr-p-c-size (**t**, get (*n*, *expr-list*))
+ lr-p-c-size-list (*n* − 1, *expr-list*)
else lr-p-c-size-list (length (*expr-list*) − 1, *expr-list*) **endif**

```
;; LR-P-PC-1 returns the number of Piton instructions before the start of
;; the expression denoted by POS in the compilation of EXPR.
```

DEFINITION:
lr-p-pc-1 (*expr*, *pos*)
= **if** ¬ listp (*pos*) **then** 0
elseif ¬ listp (*expr*) **then** 0
elseif car (*pos*) ≃ 0 **then** 0
elseif car (*expr*) = 'if
then if car (*pos*) ≃ 0 **then** 0
elseif car (*pos*) = 1 **then** lr-p-pc-1 (cadr (*expr*), cdr (*pos*))
elseif car (*pos*) = 2
then 3
+ lr-p-c-size (**t**, cadr (*expr*))
+ lr-p-pc-1 (caddr (*expr*), cdr (*pos*))
else lr-p-c-size (**t**, cadr (*expr*))
+ lr-p-c-size (**t**, caddr (*expr*))
+ lr-p-pc-1 (cadddr (*expr*), cdr (*pos*))
+ 4 **endif**
elseif car (*expr*) = S-TEMP-FETCH **then** 0
elseif car (*expr*) = S-TEMP-EVAL **then** lr-p-pc-1 (cadr (*expr*), cdr (*pos*))
elseif car (*expr*) = S-TEMP-TEST
then lr-p-pc-1 (cadr (*expr*), cdr (*pos*)) + 4
elseif car (*expr*) = 'quote **then** 0
else lr-p-c-size-list (car (*pos*) − 1, *expr*)
+ lr-p-pc-1 (get (car (*pos*), *expr*), cdr (*pos*)) **endif**

DEFINITION:
lr-p-pc (*l*)
= tag ('pc,
cons (area-name (p-pc (*l*)),
lr-p-pc-1 (program-body (p-current-program (*l*)), offset (p-pc (*l*)))))

EVENT: Disable lr-p-pc.

```
DEFINITION:
s->lr1 (s, l, table)
=   p-state (tag ('pc, cons (s-pname (s), s-pos (s))),
             p-ctrl-stk (l),
             p-temp-stk (l),
             lr-compile-programs (s-progs (s), table),
             p-data-segment (l),
             p-max-ctrl-stk-size (l),
             p-max-temp-stk-size (l),
             p-word-size (l),
             s-err-flag (s))
```

EVENT: Disable s->lr1.

```
;; Returns an P-STATE.
;; FREE-HEAP-SIZE is number of free nodes in resulting P-STATE.

DEFINITION:
s->lr (s, fheap-size, max-ctrl, max-temp, word-size)
=   let temp-alist be lr-make-temp-name-alist (strip-cars (s-temps (s)),
                                               strip-cars (s-params (s))),
        dataseg-table be lr-data-seg-table (s-progs (s),
                                            s-params (s),
                                            fheap-size)
    in
    let return-pc be tag ('pc,
                          cons (s-pname (s),
                                lr-p-pc-1 (lr-compile-body (t,
                                                            s-body (s-prog (s)),
                                                            temp-alist,
                                                            cdr (dataseg-table)),
                                           s-pos (s))))
    in
    s->lr1 (s,
            p-state (nil,
                     lr-initial-cstk (s-params (s),
                                      temp-alist,
                                      cdr (dataseg-table),
                                      return-pc),
                     nil,
                     nil,
```

car (*dataseg-table*),
max-ctrl,
max-temp,
word-size,
nil),
cdr (*dataseg-table*)) **endlet endlet**

EVENT: Disable s->lr.

DEFINITION:
lr-params (*frame*, *p*)
= firstn (length (formal-vars (p-current-program (*p*))), bindings (*frame*))

EVENT: Disable lr-params.

DEFINITION:
lr-temps (*frame*, *p*)
= restn (length (formal-vars (p-current-program (*p*))), bindings (*frame*))

EVENT: Disable lr-temps.

DEFINITION:
lr-set-expr (*s1*, *s2*, *pos*)
= p-state (tag ('pc, cons (area-name (p-pc (*s2*)), *pos*)),
p-ctrl-stk (*s1*),
p-temp-stk (*s1*),
p-prog-segment (*s2*),
p-data-segment (*s1*),
p-max-ctrl-stk-size (*s1*),
p-max-temp-stk-size (*s1*),
p-word-size (*s1*),
p-psw (*s1*))

DEFINITION:
lr-set-error (*s*, *flag*)
= p-state (p-pc (*s*),
p-ctrl-stk (*s*),
p-temp-stk (*s*),
p-prog-segment (*s*),
p-data-segment (*s*),
p-max-ctrl-stk-size (*s*),
p-max-temp-stk-size (*s*),
p-word-size (*s*),
flag)

DEFINITION:
lr-set-pos (s, *pos*)
= p-state (tag (`'pc`, cons (area-name (p-pc (s)), *pos*)),
p-ctrl-stk (s),
p-temp-stk (s),
p-prog-segment (s),
p-data-segment (s),
p-max-ctrl-stk-size (s),
p-max-temp-stk-size (s),
p-word-size (s),
p-psw (s))

DEFINITION:
lr-set-tstk (s, *temp-stk*)
= p-state (p-pc (s),
p-ctrl-stk (s),
temp-stk,
p-prog-segment (s),
p-data-segment (s),
p-max-ctrl-stk-size (s),
p-max-temp-stk-size (s),
p-word-size (s),
p-psw (s))

DEFINITION:
lr-pop-tstk (s)
= **if** p-psw (s) = `'run`
then if listp (p-temp-stk (s)) **then** lr-set-tstk (s, pop (p-temp-stk (s)))
else lr-set-error (s, `'lr-pop-tstk-empty-stack`) **endif**
else s **endif**

DEFINITION:
lr-push-tstk (s, *value*)
= **if** p-psw (s) = `'run`
then if length (p-temp-stk (s)) < p-max-temp-stk-size (s)
then lr-set-tstk (s, push (*value*, p-temp-stk (s)))
else lr-set-error (s, `'lr-push-tstk-full-stack`) **endif**
else s **endif**

EVENT: Disable lr-push-tstk.

DEFINITION:
lr-if-ok (l)
= **if** p-max-temp-stk-size (l) $\not<$ (`1` + length (p-temp-stk (l))) **then** l
else lr-set-error (l, `'if-temp-stk-overflow`) **endif**

EVENT: Disable lr-if-ok.

DEFINITION:
lr-set-temp (s, *value*, *var-name*)
= **if** p-psw (s) = `'run`
then p-state (p-pc (s),
set-local-var-value (*value*, *var-name*, p-ctrl-stk (s)),
p-temp-stk (s),
p-prog-segment (s),
p-data-segment (s),
p-max-ctrl-stk-size (s),
p-max-temp-stk-size (s),
p-word-size (s),
p-psw (s))
else s **endif**

EVENT: Disable lr-set-temp.

DEFINITION:
lr-eval-temp-setp (s)
= (local-var-value (caddr (lr-expr (s)), p-ctrl-stk (s)) $\neq$ LR-UNDEF-ADDR)

EVENT: Disable lr-eval-temp-setp.

DEFINITION:
lr-do-temp-fetch (s)
= **if** lr-eval-temp-setp (s)
then lr-push-tstk (s, local-var-value (caddr (lr-expr (s)), p-ctrl-stk (s)))
else lr-set-error (s, `'temp-fetch-not-set`) **endif**

EVENT: Disable lr-do-temp-fetch.

DEFINITION:
lr-pop-cstk (s)
= **if** p-psw (s) = `'run`
then p-state (p-pc (s),
pop (p-ctrl-stk (s)),
p-temp-stk (s),
p-prog-segment (s),
p-data-segment (s),
p-max-ctrl-stk-size (s),
p-max-temp-stk-size (s),

p-word-size (s),
p-psw (s))
else s **endif**

EVENT: Disable lr-pop-cstk.

DEFINITION:
lr-type-contents-p ($object$, tag, $contents$)
= ((type ($object$) = tag) ∧ (untag ($object$) = $contents$))

```
;; The following functions are used for the Piton code and to compute the LR
;; value for certain classes of functions (e.g. all shell accessors).

;; NOTE: The 'clock' functions get a Piton state.  This is the state just
;; BEFORE the execution of the appropriate CALL instruction.  Therefore
;; to look at the parameters, it is necessary to look at the temp stack.
;; The clock function return the number of Piton instructions necessary to
;; run the CALL and the code for the SUBR.

;; Recognizers
```

DEFINITION:
p-recognizer-code ($name$, tag)
= list ($name$,
'nil,
'nil,
'(fetch),
list ('push-constant, tag ('nat, tag)),
'(eq),
'(test-bool-and-jump f false),
list ('push-constant, LR-T-ADDR),
'(ret),
list ('dl, 'false, 'nil, list ('push-constant, LR-F-ADDR)),
'(ret))

DEFINITION: p-recognizer-clock (p-$state$, tag) = 7

```
;; Accessor
```

DEFINITION:
p-accessor-code ($name$, tag, $default$, $offset$)
= list ($name$,
'(x),
'nil,
'(push-local x),

```
        '(fetch),
        list('push-constant, tag('nat, tag)),
        '(eq),
        '(test-bool-and-jump t arg1),
        list('push-constant, default),
        '(ret),
        '(dl arg1 nil (push-local x)),
        list('push-constant, tag('nat, offset)),
        '(add-addr),
        '(fetch),
        '(ret))
```

DEFINITION:
p-accessor-clock(p, tag)
= **if** fetch(top(p-temp-stk(p)), p-data-segment(p)) = tag('nat, tag)
  then 11
  else 8 **endif**

```
;; Now comes the actual code and values
```

DEFINITION:
P-CAR-CODE = p-accessor-code('car, LR-CONS-TAG, LR-0-ADDR, LR-CAR-OFFSET)

DEFINITION: p-car-clock(p) = p-accessor-clock(p, LR-CONS-TAG)

EVENT: Disable p-car-clock.

DEFINITION:
P-CDR-CODE = p-accessor-code('cdr, LR-CONS-TAG, LR-0-ADDR, LR-CDR-OFFSET)

DEFINITION: p-cdr-clock(p) = p-accessor-clock(p, LR-CONS-TAG)

EVENT: Disable p-cdr-clock.

DEFINITION:
P-CONS-CODE
=
```
  list('cons,
       'nil,
       '((temp (nat 0))),
       '(push-global free-ptr),
       list('push-constant, tag('nat, LR-CDR-OFFSET)),
       '(add-addr),
       '(deposit),
```

```
        '(push-global free-ptr),
        list('push-constant, tag('nat, LR-CAR-OFFSET)),
        '(add-addr),
        '(deposit),
        '(push-global free-ptr),
        '(push-global free-ptr),
        list('push-constant, tag('nat, LR-REF-COUNT-OFFSET)),
        '(add-addr),
        '(set-local temp),
        '(fetch),
        '(push-constant (nat 1)),
        '(push-local temp),
        '(deposit),
        list('push-constant, tag('nat, LR-CONS-TAG)),
        '(push-global free-ptr),
        '(deposit),
        '(pop-global free-ptr),
        '(ret))
```

DEFINITION: p-cons-clock(p) = 23

EVENT: Disable p-cons-clock.

```
DEFINITION:
P-FALSE-CODE
=   list('false,
        'nil,
        'nil,
        list('push-constant, LR-F-ADDR),
        '(ret))
```

DEFINITION: p-false-clock(p) = 3

EVENT: Disable p-false-clock.

```
;; FALSEP TAKES ONE IMPLICIT ARG ON STACK.

DEFINITION:
P-FALSEP-CODE
=   list('falsep,
        'nil,
        'nil,
        list('push-constant, LR-F-ADDR),
        '(eq),
```

```
        '(test-bool-and-jump t true),
        list ('push-constant, LR-F-ADDR),
        '(ret),
        list ('dl, 'true, 'nil, list ('push-constant, LR-T-ADDR)),
        '(ret))
```

DEFINITION: p-falsep-clock (p) = 6

EVENT: Disable p-falsep-clock.

```
;; Takes an implicit arg
```

DEFINITION:
P-LISTP-CODE = p-recognizer-code ('listp, LR-CONS-TAG)

DEFINITION:
p-listp-clock (p) = p-recognizer-clock $(p$, LR-CONS-TAG)

EVENT: Disable p-listp-clock.

DEFINITION:
P-NLISTP-CODE
= list ('nlistp,
        'nil,
        'nil,
        '(fetch),
        list ('push-constant, tag ('nat, LR-CONS-TAG)),
        '(eq),
        '(test-bool-and-jump f true),
        list ('push-constant, LR-F-ADDR),
        '(ret),
        list ('dl, 'true, 'nil, list ('push-constant, LR-T-ADDR)),
        '(ret))

DEFINITION: p-nlistp-clock (p) = 7

EVENT: Disable p-nlistp-clock.

DEFINITION:
P-TRUE-CODE
= list ('true, 'nil, 'nil, list ('push-constant, LR-T-ADDR), '(ret))

DEFINITION: p-true-clock (p) = 3

EVENT: Disable p-true-clock.

```
;; The old code for TRUEP is shown below.  I used to ensure that there was
;; only one occurence of TRUE in the data-segment [namely at address
;; (lr-t-addr)], however only TRUEP took advantage of this.  LR-PROPER-HEAPP
;; has been changed to not require only one occurrence, although only one
;; should appear.  However this means we actually have to test the tag, a
;; small performance penalty for some simplicity and freedom in the spec.
```

DEFINITION:
P-TRUEP-CODE = p-recognizer-code ('truep, LR-TRUE-TAG)

```
;(defn P-TRUEP-CODE ()
;  (list 'truep '() '()
; (list 'PUSH-CONSTANT (lr-t-addr))
; '(EQ)
; '(TEST-BOOL-AND-JUMP T TRUE)
; (list 'PUSH-CONSTANT (lr-f-addr))
; '(RET)
; (list 'DL 'TRUE '() (list 'PUSH-CONSTANT (lr-t-addr)))
; '(RET)))
```

DEFINITION:
p-truep-clock (p) = p-recognizer-clock (p, LR-FALSE-TAG)

```
;(defn P-TRUEP-CLOCK (p) 6)
```

EVENT: Disable p-truep-clock.

DEFINITION:
P-RUNTIME-SUPPORT-PROGRAMS
= list (P-CAR-CODE,
P-CDR-CODE,
P-CONS-CODE,
P-FALSE-CODE,
P-FALSEP-CODE,
P-LISTP-CODE,
P-NLISTP-CODE,
P-TRUE-CODE,
P-TRUEP-CODE)

EVENT: Disable p-runtime-support-programs.

DEFINITION:
lr-convert-digit-to-ascii (*digit*) = (ASCII-0 + *digit*)

DEFINITION:
lr-convert-num-to-ascii (*number*, *list*)
= **if** *number* < 10 **then** cons (lr-convert-digit-to-ascii (*number*), *list*)
else lr-convert-num-to-ascii (*number* ÷ 10,
cons (lr-convert-digit-to-ascii (*number* **mod** 10),
list)) **endif**

DEFINITION:
lr-make-label (*n*)
= pack (cons (car (unpack ('1')),
cons (ASCII-DASH, append (lr-convert-num-to-ascii (*n*, **nil**), 0))))

EVENT: Disable lr-make-label.

DEFINITION:
label-instrs (*instrs*, *n*)
= **if** listp (*instrs*)
then cons (dl (lr-make-label (*n*), **nil**, car (*instrs*)),
label-instrs (cdr (*instrs*), 1 + *n*))
else nil endif

DEFINITION:
comp-temp-test (*expr*, *instrs*, *n*)
= append (list (list ('push-local, caddr (*expr*)),
list ('push-constant, LR-UNDEF-ADDR),
'(eq),
list ('test-bool-and-jump,
'f,
lr-make-label (*n* + 6 + length (*instrs*)))),
append (*instrs*,
list (list ('set-local, caddr (*expr*)),
list ('jump,
lr-make-label (*n* + 7 + length (*instrs*))),
list ('push-local, caddr (*expr*)))))

DEFINITION:
comp-if (*test-instrs*, *then-instrs*, *else-instrs*, *n*)
= append (*test-instrs*,
append (list (list ('push-constant, LR-F-ADDR),
'(eq),
list ('test-bool-and-jump,

```
                    't,
                    lr-make-label (n
                                   +  4
                                   +  length (test-instrs)
                                   +  length (then-instrs)))),
            append (then-instrs,
                    cons (list ('jump,
                                lr-make-label (n
                                               +  4
                                               +  length (test-instrs)
                                               +  length (then-instrs)
                                               +  length (else-instrs))),
                          else-instrs))))
```

```
;; COMP-BODY-1 returns a list of Piton instructions to compile EXPR.
;; N is the number of Piton instructions previously generated, it is used
;; to generate unique labels.
```

DEFINITION:
comp-body-1 (*flag*, *expr*, *n*)
= **if** *flag* = 'list
then if listp (*expr*)
then append (comp-body-1 (**t**, car (*expr*), *n*),
comp-body-1 ('list,
cdr (*expr*),
n + lr-p-c-size (**t**, car (*expr*))))
else nil endif
elseif listp (*expr*)
then if car (*expr*) = 'if
then comp-if (comp-body-1 (**t**, cadr (*expr*), *n*),
comp-body-1 (**t**,
caddr (*expr*),
n + 3 + lr-p-c-size (**t**, cadr (*expr*))),
comp-body-1 (**t**,
cadddr (*expr*),
n
+ 4
+ lr-p-c-size (**t**, cadr (*expr*))
+ lr-p-c-size (**t**, caddr (*expr*))),
n)
elseif car (*expr*) = S-TEMP-FETCH
then list (list ('push-local, caddr (*expr*)))
elseif car (*expr*) = S-TEMP-EVAL
then append (comp-body-1 (**t**, cadr (*expr*), *n*),

```
                    list (list ('set-local, caddr (expr))))
        elseif car (expr) = S-TEMP-TEST
        then comp-temp-test (expr, comp-body-1 (t, cadr (expr), n + 4), n)
        elseif car (expr) = 'quote
        then list (list ('push-constant, cadr (expr)))
        else append (comp-body-1 ('list, cdr (expr), n),
                     if definedp (car (expr),
                                  P-RUNTIME-SUPPORT-PROGRAMS)
                     then list (list ('call, car (expr)))
                     else list (list ('call,
                                      user-fname (car (expr)))) endif) endif
    else list (list ('push-local, expr)) endif
```

EVENT: Disable comp-body-1.

```
DEFINITION:
comp-body (body)
=   label-instrs (append (comp-body-1 (t, body, 0), '((ret))), 0)
```

EVENT: Disable comp-body.

```
DEFINITION:
comp-programs-1 (programs)
=   if listp (programs)
    then cons (lr-make-program (name (car (programs)),
                                formal-vars (car (programs)),
                                temp-var-dcls (car (programs)),
                                comp-body (program-body (car (programs)))),
               comp-programs-1 (cdr (programs)))
    else nil endif
```

```
DEFINITION:
comp-programs (programs)
=   cons (lr-make-program (name (car (programs)),
                          formal-vars (car (programs)),
                          temp-var-dcls (car (programs)),
                          label-instrs (append (comp-body-1 (t,
                                                             program-body (car (programs)),
                                                             0),
                                                list (list ('set-global,
                                                            area-name (LR-ANSWER-ADDR)),
                                                      '(ret))),
                                        0)),
         append (comp-programs-1 (cdr (programs)), P-RUNTIME-SUPPORT-PROGRAMS))
```

EVENT: Disable comp-programs.

DEFINITION:
lr-proper-exprp (*flag*, *expr*, *pnames*, *formals*, *temps*, *table*)
= **if** *flag* = `'list`
then if listp (*expr*)
then lr-proper-exprp (**t**, car (*expr*), *pnames*, *formals*, *temps*, *table*)
∧ lr-proper-exprp (`'list`,
cdr (*expr*),
pnames,
formals,
temps,
table)
else *expr* = **nil endif**
elseif litatom (*expr*) **then** *expr* ∈ *formals*
elseif *expr* ≃ **nil then f**
elseif ¬ plistp (*expr*) **then f**
elseif car (*expr*) = S-TEMP-FETCH
then (caddr (*expr*) ∈ *temps*) ∧ (length (*expr*) = `3`)
elseif (car (*expr*) = S-TEMP-EVAL) ∨ (car (*expr*) = S-TEMP-TEST)
then (caddr (*expr*) ∈ *temps*)
∧ (length (*expr*) = `3`)
∧ lr-proper-exprp (**t**, cadr (*expr*), *pnames*, *formals*, *temps*, *table*)
elseif car (*expr*) = `'quote`
then (type (cadr (*expr*)) = `'addr`)
∧ (cadr (*expr*) ∈ strip-cdrs (*table*))
∧ (length (cdr (*expr*)) = arity (car (*expr*)))
elseif subrp (car (*expr*))
then (length (cdr (*expr*)) = arity (car (*expr*)))
∧ ((car (*expr*) = `'if`)
∨ definedp (car (*expr*), P-RUNTIME-SUPPORT-PROGRAMS))
∧ (car (*expr*) ∉ *pnames*)
∧ lr-proper-exprp (`'list`,
cdr (*expr*),
pnames,
formals,
temps,
table)
elseif body (car (*expr*))
then (length (cdr (*expr*)) = arity (car (*expr*)))
∧ (car (*expr*) ∈ *pnames*)
∧ lr-proper-exprp (`'list`,
cdr (*expr*),

pnames,
formals,
temps,
table)
else f endif

DEFINITION:
all-undef-addrs (*list*)
= **if** listp (*list*)
then (car (*list*) = LR-UNDEF-ADDR) ∧ all-undef-addrs (cdr (*list*))
else t endif

DEFINITION:
lr-programs-properp-1 (*programs*, *program-names*, *table*)
= **if** listp (*programs*)
then all-litatoms (formal-vars (car (*programs*)))
∧ all-litatoms (strip-cars (temp-var-dcls (car (*programs*))))
∧ all-undef-addrs (strip-cadrs (temp-var-dcls (car (*programs*))))
∧ lr-proper-exprp (**t**,
program-body (car (*programs*)),
program-names,
formal-vars (car (*programs*)),
strip-cars (temp-var-dcls (car (*programs*))),
table)
∧ lr-programs-properp-1 (cdr (*programs*), *program-names*, *table*)
else t endif

EVENT: Disable lr-programs-properp-1.

DEFINITION:
lr-programs-properp (*l*, *table*)
= (definedp (area-name (p-pc (*l*)), p-prog-segment (*l*))
∧ (caar (p-prog-segment (*l*)) = 'main)
∧ all-user-fnamesp (cdr (strip-cars (p-prog-segment (*l*))))
∧ lr-programs-properp-1 (p-prog-segment (*l*),
strip-logic-fnames (cdr (p-prog-segment (*l*))),
table))

EVENT: Disable lr-programs-properp.

THEOREM: lr-p-c-size-flag-not-list-not-0
(*flag* ≠ 'list) → (lr-p-c-size (*flag*, *expr*) ≠ 0)

THEOREM: difference-decreases
((*x* ≮ *y*) ∧ (*y* ≠ 0)) → (((*x* − *y*) < *x*) = **t**)

DEFINITION:
lr->p (*p*)
= p-state (lr-p-pc (*p*),
p-ctrl-stk (*p*),
p-temp-stk (*p*),
comp-programs (p-prog-segment (*p*)),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
p-psw (*p*))

EVENT: Disable lr->p.

DEFINITION:
p-set-pc (*p*, *pc*)
= p-state (*pc*,
p-ctrl-stk (*p*),
p-temp-stk (*p*),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
p-psw (*p*))

```
;; It should be the case that (P-CURRENT-INSTRUCTION p) = (CALL subr)
;; therefore we need to run P one more step than the clock functions
;; below to do the CALL.
```

DEFINITION:
p-run-subr (*subr*, *p*)
= **case on** *subr*:
case = *car*
then p (*p*, p-car-clock (*p*))
case = *cdr*
then p (*p*, p-cdr-clock (*p*))
case = *cons*
then p (*p*, p-cons-clock (*p*))
case = *false*
then p (*p*, p-false-clock (*p*))
case = *falsep*
then p (*p*, p-falsep-clock (*p*))
case = *listp*

```
    then p (p, p-listp-clock (p))
  case = nlistp
    then p (p, p-nlistp-clock (p))
  case = true
    then p (p, p-true-clock (p))
  case = truep
    then p (p, p-truep-clock (p))
  otherwise p-halt (p, 'bad-subr) endcase
```

EVENT: Disable p-run-subr.

DEFINITION:

```
lr-return-pc (l)
=  add-addr (lr-p-pc (l), lr-p-c-size ('list, cdr (lr-expr (l))))
```

EVENT: Disable lr-return-pc.

DEFINITION:

```
lr-apply-subr (l, new-l)
=  let res be p-run-subr (car (lr-expr (l)),
                          p-set-pc (lr->p (new-l), lr-return-pc (l)))
   in
   p-state (p-pc (new-l),
            p-ctrl-stk (res),
            p-temp-stk (res),
            p-prog-segment (new-l),
            p-data-segment (res),
            p-max-ctrl-stk-size (res),
            p-max-temp-stk-size (res),
            p-word-size (res),
            p-psw (res)) endlet
```

EVENT: Disable lr-apply-subr.

DEFINITION:

```
lr-funcall (l, new-l)
=  let prog be definition (user-fname (car (lr-expr (l))),
                           p-prog-segment (l)),
       newest-l be p-set-pc (lr->p (new-l), lr-return-pc (l))
   in
   if p-call-okp (list ('call, user-fname (car (lr-expr (l)))),
                  newest-l)
   then p-state (tag ('pc, cons (user-fname (car (lr-expr (l))), nil)),
```

push (make-p-call-frame (formal-vars (*prog*),
p-temp-stk (*new-l*),
temp-var-dcls (*prog*),
add1-addr (p-pc (*newest-l*))),
p-ctrl-stk (*new-l*)),
popn (length (formal-vars (*prog*)), p-temp-stk (*new-l*)),
p-prog-segment (*new-l*),
p-data-segment (*new-l*),
p-max-ctrl-stk-size (*new-l*),
p-max-temp-stk-size (*new-l*),
p-word-size (*new-l*),
`'run`)
else p-halt (*new-l*, x-y-error-msg (`'p`, `'call`)) **endif endlet**

EVENT: Disable lr-funcall.

```
;; The following lemmas are needed to admit LR-EVAL
```

THEOREM: p-accessors-lr-set-expr
(p-pc (lr-set-expr (*s1*, *s2*, *pos*)) = tag (`'pc`, cons (area-name (p-pc (*s2*)), *pos*)))
∧ (p-ctrl-stk (lr-set-expr (*s1*, *s2*, *pos*)) = p-ctrl-stk (*s1*))
∧ (p-temp-stk (lr-set-expr (*s1*, *s2*, *pos*)) = p-temp-stk (*s1*))
∧ (p-prog-segment (lr-set-expr (*s1*, *s2*, *pos*)) = p-prog-segment (*s2*))
∧ (p-data-segment (lr-set-expr (*s1*, *s2*, *pos*)) = p-data-segment (*s1*))
∧ (p-max-ctrl-stk-size (lr-set-expr (*s1*, *s2*, *pos*))
= p-max-ctrl-stk-size (*s1*))
∧ (p-max-temp-stk-size (lr-set-expr (*s1*, *s2*, *pos*))
= p-max-temp-stk-size (*s1*))
∧ (p-word-size (lr-set-expr (*s1*, *s2*, *pos*)) = p-word-size (*s1*))
∧ (p-psw (lr-set-expr (*s1*, *s2*, *pos*)) = p-psw (*s1*))

EVENT: Disable lr-set-expr.

THEOREM: p-accessors-lr-set-tstk
(p-pc (lr-set-tstk (*s*, *ts*)) = p-pc (*s*))
∧ (p-ctrl-stk (lr-set-tstk (*s*, *ts*)) = p-ctrl-stk (*s*))
∧ (p-temp-stk (lr-set-tstk (*s*, *ts*)) = *ts*)
∧ (p-prog-segment (lr-set-tstk (*s*, *ts*)) = p-prog-segment (*s*))
∧ (p-data-segment (lr-set-tstk (*s*, *ts*)) = p-data-segment (*s*))
∧ (p-max-ctrl-stk-size (lr-set-tstk (*s*, *ts*)) = p-max-ctrl-stk-size (*s*))
∧ (p-max-temp-stk-size (lr-set-tstk (*s*, *ts*)) = p-max-temp-stk-size (*s*))
∧ (p-word-size (lr-set-tstk (*s*, *ts*)) = p-word-size (*s*))
∧ (p-psw (lr-set-tstk (*s*, *ts*)) = p-psw (*s*))

EVENT: Disable lr-set-tstk.

THEOREM: p-accessors-lr-set-error
(p-pc (lr-set-error (s, *flag*)) = p-pc (s))
∧ (p-ctrl-stk (lr-set-error (s, *flag*)) = p-ctrl-stk (s))
∧ (p-temp-stk (lr-set-error (s, *flag*)) = p-temp-stk (s))
∧ (p-prog-segment (lr-set-error (s, *flag*)) = p-prog-segment (s))
∧ (p-data-segment (lr-set-error (s, *flag*)) = p-data-segment (s))
∧ (p-max-ctrl-stk-size (lr-set-error (s, *flag*)) = p-max-ctrl-stk-size (s))
∧ (p-max-temp-stk-size (lr-set-error (s, *flag*)) = p-max-temp-stk-size (s))
∧ (p-word-size (lr-set-error (s, *flag*)) = p-word-size (s))
∧ (p-psw (lr-set-error (s, *flag*)) = *flag*)

EVENT: Disable lr-set-error.

THEOREM: p-accessors-lr-set-pos
(p-pc (lr-set-pos (s, *pos*)) = tag ('pc, cons (area-name (p-pc (s)), *pos*)))
∧ (p-ctrl-stk (lr-set-pos (s, *pos*)) = p-ctrl-stk (s))
∧ (p-temp-stk (lr-set-pos (s, *pos*)) = p-temp-stk (s))
∧ (p-prog-segment (lr-set-pos (s, *pos*)) = p-prog-segment (s))
∧ (p-data-segment (lr-set-pos (s, *pos*)) = p-data-segment (s))
∧ (p-max-ctrl-stk-size (lr-set-pos (s, *pos*)) = p-max-ctrl-stk-size (s))
∧ (p-max-temp-stk-size (lr-set-pos (s, *pos*)) = p-max-temp-stk-size (s))
∧ (p-word-size (lr-set-pos (s, *pos*)) = p-word-size (s))
∧ (p-psw (lr-set-pos (s, *pos*)) = p-psw (s))

EVENT: Disable lr-set-pos.

THEOREM: p-accessors-lr-pop-tstk
(p-pc (lr-pop-tstk (s)) = p-pc (s))
∧ (p-ctrl-stk (lr-pop-tstk (s)) = p-ctrl-stk (s))
∧ (p-prog-segment (lr-pop-tstk (s)) = p-prog-segment (s))
∧ (p-data-segment (lr-pop-tstk (s)) = p-data-segment (s))
∧ (p-max-ctrl-stk-size (lr-pop-tstk (s)) = p-max-ctrl-stk-size (s))
∧ (p-max-temp-stk-size (lr-pop-tstk (s)) = p-max-temp-stk-size (s))
∧ (p-word-size (lr-pop-tstk (s)) = p-word-size (s))

THEOREM: p-temp-stk-lr-pop-tstk
p-temp-stk (lr-pop-tstk (s))
= **if** listp (p-temp-stk (s)) ∧ (p-psw (s) = 'run)
then pop (p-temp-stk (s))
else p-temp-stk (s) **endif**

EVENT: Disable lr-pop-tstk.

THEOREM: area-name-tag
area-name (tag (*tag*, *adp*)) = adp-name (*adp*)

THEOREM: offset-tag
offset (tag (*tag*, *adp*)) = adp-offset (*adp*)

THEOREM: p-current-program-lr-set-expr
p-current-program (lr-set-expr (*s1*, *s2*, *pos*)) = p-current-program (*s2*)

THEOREM: p-current-program-lr-set-pos
p-current-program (lr-set-pos (*s*, *pos*)) = p-current-program (*s*)

THEOREM: lr-expr-lr-set-expr
lr-expr (lr-set-expr (*s1*, *s2*, dv (offset (p-pc (*s2*)), *n*))) = get (*n*, lr-expr (*s2*))

THEOREM: lr-expr-lr-set-pos-t
lr-expr (lr-set-pos (*s*, dv (offset (p-pc (*s*)), *n*))) = get (*n*, lr-expr (*s*))

THEOREM: lr-expr-flag-list-car
listp (offset (p-pc (*p*))) → (car (lr-expr-list (*p*)) = lr-expr (*p*))

THEOREM: number-cons-lr-expr-t-list
(listp (lr-expr-list (*p*)) ∧ listp (offset (p-pc (*p*))))
→ (number-cons (lr-expr (*p*)) < number-cons (lr-expr-list (*p*)))

THEOREM: lr-expr-lr-set-expr-nx
(listp (offset (p-pc (*p*))) ∧ listp (lr-expr-list (*p*)))
→ (lr-expr-list (lr-set-expr (*p1*, *p*, nx (offset (p-pc (*p*)))))
= cdr (lr-expr-list (*p*)))

THEOREM: lr-expr-list-lr-set-pos-dv-1
listp (lr-expr (*p*))
→ (lr-expr-list (lr-set-pos (*p*, dv (offset (p-pc (*p*)), 1)))
= cdr (lr-expr (*p*)))

```
;; If FLAG is 'LIST then state contains a list of expressions,
;; otherwise it is just one.
;; Returns a P-STATE.  The result is left on the temp stack.
;; If the error flag of the resulting state is 'HALT then we terminated
;; normally.  If the flag is 'RUN we have not terminated yet.
;; If the flag is anything else we got an error.
```

DEFINITION:
lr-eval (*flag*, *l*, *c*)
= **if** p-psw (*l*) ≠ `'run` **then** *l*
elseif *flag* = `'list`

```
then if offset (p-pc (l)) ≃ nil
     then lr-set-error (l, 'bad-list-position)
     elseif listp (lr-expr-list (l))
     then lr-eval ('list,
                   lr-set-expr (lr-eval (t, l, c), l, nx (offset (p-pc (l)))),
                   c)
     else l endif
elseif c ≃ 0 then lr-set-error (l, 'out-of-time)
elseif litatom (lr-expr (l))
then lr-push-tstk (l, local-var-value (lr-expr (l), p-ctrl-stk (l)))
elseif lr-expr (l) ≃ nil then lr-set-error (l, 'bad-expression)
elseif car (lr-expr (l)) = 'if
then let test be lr-if-ok (lr-eval (t,
                                   lr-set-pos (l,
                                               dv (offset (p-pc (l)), 1)),
                                   c))
     in
     if p-psw (test) = 'run
     then if top (p-temp-stk (test)) ≠ LR-F-ADDR
          then lr-eval (t,
                        lr-set-expr (lr-pop-tstk (test),
                                     l,
                                     dv (offset (p-pc (l)), 2)),
                        c)
          else lr-eval (t,
                        lr-set-expr (lr-pop-tstk (test),
                                     l,
                                     dv (offset (p-pc (l)), 3)),
                        c) endif
     else test endif endlet
elseif car (lr-expr (l)) = S-TEMP-EVAL
then let l1 be lr-eval (t, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
     in
     lr-set-temp (l1, top (p-temp-stk (l1)), caddr (lr-expr (l))) endlet
elseif car (lr-expr (l)) = S-TEMP-TEST
then let l1 be lr-eval (t, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
     in
     if p-max-temp-stk-size (l)
       ≮ (2 + length (p-temp-stk (l)))
     then if lr-eval-temp-setp (l) then lr-do-temp-fetch (l)
          else lr-set-temp (l1,
                            top (p-temp-stk (l1)),
                            caddr (lr-expr (l))) endif
     else lr-set-error (l,
```

```
                              'lr-temp-setp-temp-stack-overflow) endif endlet
    elseif car (lr-expr (l)) = S-TEMP-FETCH then lr-do-temp-fetch (l)
    elseif car (lr-expr (l)) = 'quote
    then lr-push-tstk (l, cadr (lr-expr (l)))
    elseif p-psw (lr-eval ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c))
           ≠  'run
    then lr-eval ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
    elseif subrp (car (lr-expr (l)))
    then lr-apply-subr (l,
                        lr-eval ('list,
                                 lr-set-pos (l, dv (offset (p-pc (l)), 1)),
                                 c))
    elseif litatom (car (lr-expr (l)))
    then let fs be lr-funcall (l,
                               lr-eval ('list,
                                        lr-set-pos (l,
                                                    dv (offset (p-pc (l)), 1)),
                                        c))
         in
         lr-set-expr (lr-pop-cstk (lr-eval (t, fs, c − 1)),
                      l,
                      offset (p-pc (l))) endlet
    else lr-set-error (l, 'bad-instruction) endif

;; Proper LR STATES

;; Sometimes we only need to know that LR-PROPER-P-AREASP holds on
;; a data-segment instead of LR-PROPER-P-DATA-SEGMENTP
```

DEFINITION:
lr-proper-p-areasp (*data-seg*)
= **if** *data-seg* ≃ **nil** **then** *data-seg* = **nil**
else let *area* **be** car (*data-seg*)
in
litatom (car (*area*))
∧ listp (cdr (*area*))
∧ (¬ definedp (car (*area*), cdr (*data-seg*)))
∧ lr-proper-p-areasp (cdr (*data-seg*)) **endlet endif**

```
;; First we prove that LR-EVAL preserves PROPER-P-STATEP.
```

THEOREM: p-accessors-lr-funcall
(p-prog-segment (lr-funcall (*l*, *new-l*)) = p-prog-segment (*new-l*))
∧ (p-data-segment (lr-funcall (*l*, *new-l*)) = p-data-segment (*new-l*))

$\wedge$ (p-max-ctrl-stk-size (lr-funcall (l, *new-l*))
= p-max-ctrl-stk-size (*new-l*))
$\wedge$ (p-max-temp-stk-size (lr-funcall (l, *new-l*))
= p-max-temp-stk-size (*new-l*))
$\wedge$ (p-word-size (lr-funcall (l, *new-l*)) = p-word-size (*new-l*))

THEOREM: p-accessors-lr-push-tstk
(p-pc (lr-push-tstk (s, v)) = p-pc (s))
$\wedge$ (p-ctrl-stk (lr-push-tstk (s, v)) = p-ctrl-stk (s))
$\wedge$ (p-prog-segment (lr-push-tstk (s, v)) = p-prog-segment (s))
$\wedge$ (p-data-segment (lr-push-tstk (s, v)) = p-data-segment (s))
$\wedge$ (p-max-ctrl-stk-size (lr-push-tstk (s, v)) = p-max-ctrl-stk-size (s))
$\wedge$ (p-max-temp-stk-size (lr-push-tstk (s, v)) = p-max-temp-stk-size (s))
$\wedge$ (p-word-size (lr-push-tstk (s, v)) = p-word-size (s))

THEOREM: p-accessors-lr-if-ok
(p-pc (lr-if-ok (l)) = p-pc (l))
$\wedge$ (p-ctrl-stk (lr-if-ok (l)) = p-ctrl-stk (l))
$\wedge$ (p-temp-stk (lr-if-ok (l)) = p-temp-stk (l))
$\wedge$ (p-prog-segment (lr-if-ok (l)) = p-prog-segment (l))
$\wedge$ (p-data-segment (lr-if-ok (l)) = p-data-segment (l))
$\wedge$ (p-max-ctrl-stk-size (lr-if-ok (l)) = p-max-ctrl-stk-size (l))
$\wedge$ (p-max-temp-stk-size (lr-if-ok (l)) = p-max-temp-stk-size (l))
$\wedge$ (p-word-size (lr-if-ok (l)) = p-word-size (l))

THEOREM: p-accessors-lr-set-temp
(p-pc (lr-set-temp (s, v, n)) = p-pc (s))
$\wedge$ (p-ctrl-stk (lr-set-temp (s, v, n))
= **if** p-psw (s) = `'run`
then set-local-var-value (v, n, p-ctrl-stk (s))
else p-ctrl-stk (s) **endif**)
$\wedge$ (p-temp-stk (lr-set-temp (s, v, n)) = p-temp-stk (s))
$\wedge$ (p-prog-segment (lr-set-temp (s, v, n)) = p-prog-segment (s))
$\wedge$ (p-data-segment (lr-set-temp (s, v, n)) = p-data-segment (s))
$\wedge$ (p-max-ctrl-stk-size (lr-set-temp (s, v, n)) = p-max-ctrl-stk-size (s))
$\wedge$ (p-max-temp-stk-size (lr-set-temp (s, v, n)) = p-max-temp-stk-size (s))
$\wedge$ (p-word-size (lr-set-temp (s, v, n)) = p-word-size (s))
$\wedge$ (p-psw (lr-set-temp (s, v, n)) = p-psw (s))

THEOREM: p-accessors-lr-do-temp-fetch
(p-pc (lr-do-temp-fetch (s)) = p-pc (s))
$\wedge$ (p-ctrl-stk (lr-do-temp-fetch (s)) = p-ctrl-stk (s))
$\wedge$ (p-prog-segment (lr-do-temp-fetch (s)) = p-prog-segment (s))
$\wedge$ (p-data-segment (lr-do-temp-fetch (s)) = p-data-segment (s))
$\wedge$ (p-max-ctrl-stk-size (lr-do-temp-fetch (s)) = p-max-ctrl-stk-size (s))

$\wedge$ (p-max-temp-stk-size (lr-do-temp-fetch (s)) = p-max-temp-stk-size (s))
$\wedge$ (p-word-size (lr-do-temp-fetch (s)) = p-word-size (s))

THEOREM: p-accessors-lr-pop-cstk
(p-pc (lr-pop-cstk (s)) = p-pc (s))
$\wedge$ (p-ctrl-stk (lr-pop-cstk (s))
= **if** p-psw (s) = `'run` **then** pop (p-ctrl-stk (s))
else p-ctrl-stk (s) **endif**)
$\wedge$ (p-temp-stk (lr-pop-cstk (s)) = p-temp-stk (s))
$\wedge$ (p-prog-segment (lr-pop-cstk (s)) = p-prog-segment (s))
$\wedge$ (p-data-segment (lr-pop-cstk (s)) = p-data-segment (s))
$\wedge$ (p-max-ctrl-stk-size (lr-pop-cstk (s)) = p-max-ctrl-stk-size (s))
$\wedge$ (p-max-temp-stk-size (lr-pop-cstk (s)) = p-max-temp-stk-size (s))
$\wedge$ (p-word-size (lr-pop-cstk (s)) = p-word-size (s))
$\wedge$ (p-psw (lr-pop-cstk (s)) = p-psw (s))

THEOREM: lr-eval-if-p-psw-1
((*flag* $\neq$ `'list`)
$\wedge$ (p-psw (lr-eval (*flag*, l, c)) = `'run`)
$\wedge$ (car (lr-expr (l)) = `'if`)
$\wedge$ (p-psw (l) = `'run`)
$\wedge$ ($c \not\simeq$ `0`)
$\wedge$ listp (lr-expr (l)))
$\rightarrow$ (p-psw (lr-eval (**t**, lr-set-pos (l, dv (offset (p-pc (l)), `1`)), c)) = `'run`)

EVENT: Disable lr-eval-if-p-psw-1.

THEOREM: adp-offset-untag-add-addr
adp-offset (untag (add-addr (*addr*, n))) = (offset (*addr*) + n)

THEOREM: adp-offset-untag-sub-addr
adp-offset (untag (sub-addr (*addr*, n))) = (offset (*addr*) $-$ n)

THEOREM: adp-name-untag-sub-addr
adp-name (untag (sub-addr (*addr*, n))) = adp-name (untag (*addr*))

THEOREM: adp-offset-cons
adp-offset (cons (*area-name*, *offset*)) = *offset*

THEOREM: p-accessors-lr->p
(p-pc (lr->p (l)) = lr-p-pc (l))
$\wedge$ (p-ctrl-stk (lr->p (l)) = p-ctrl-stk (l))
$\wedge$ (p-temp-stk (lr->p (l)) = p-temp-stk (l))
$\wedge$ (p-prog-segment (lr->p (l)) = comp-programs (p-prog-segment (l)))
$\wedge$ (p-data-segment (lr->p (l)) = p-data-segment (l))

$\wedge$ (p-max-ctrl-stk-size (lr->p (l)) = p-max-ctrl-stk-size (l))
$\wedge$ (p-max-temp-stk-size (lr->p (l)) = p-max-temp-stk-size (l))
$\wedge$ (p-word-size (lr->p (l)) = p-word-size (l))
$\wedge$ (p-psw (lr->p (l)) = p-psw (l))

THEOREM: type-lr-p-pc
type (lr-p-pc (l)) = 'pc

THEOREM: cddr-nil-lr-p-pc
cddr (lr-p-pc (l)) = **nil**

THEOREM: listp-untag-lr-p-pc
listp (untag (lr-p-pc (l)))

THEOREM: numberp-cdr-lr-p-pc
cdr (untag (lr-p-pc (l))) $\in$ **N**

THEOREM: car-untag-lr-p-pc
car (untag (lr-p-pc (p))) = car (untag (p-pc (p)))

THEOREM: area-name-lr-p-pc
area-name (lr-p-pc (p)) = area-name (p-pc (p))

THEOREM: definedp-comp-programs-1-definedp-orig
definedp (x, comp-programs-1 (*programs*)) = definedp (x, *programs*)

THEOREM: definedp-append
definedp (x, append (*l1*, *l2*)) = (definedp (x, *l1*) $\vee$ definedp (x, *l2*))

THEOREM: definedp-comp-programs-definedp-orig
definedp (x, *programs*) $\rightarrow$ definedp (x, comp-programs (*programs*))

THEOREM: p-accessors-p-halt
(p-pc (p-halt (p, *psw*)) = p-pc (p))
$\wedge$ (p-ctrl-stk (p-halt (p, *psw*)) = p-ctrl-stk (p))
$\wedge$ (p-temp-stk (p-halt (p, *psw*)) = p-temp-stk (p))
$\wedge$ (p-prog-segment (p-halt (p, *psw*)) = p-prog-segment (p))
$\wedge$ (p-data-segment (p-halt (p, *psw*)) = p-data-segment (p))
$\wedge$ (p-max-ctrl-stk-size (p-halt (p, *psw*)) = p-max-ctrl-stk-size (p))
$\wedge$ (p-max-temp-stk-size (p-halt (p, *psw*)) = p-max-temp-stk-size (p))
$\wedge$ (p-word-size (p-halt (p, *psw*)) = p-word-size (p))
$\wedge$ (p-psw (p-halt (p, *psw*)) = *psw*)

EVENT: Disable p-halt.

THEOREM: p-accessors-p-set-pc
(p-pc (p-set-pc (*p*, *pc*)) = *pc*)
∧ (p-ctrl-stk (p-set-pc (*p*, *pc*)) = p-ctrl-stk (*p*))
∧ (p-temp-stk (p-set-pc (*p*, *pc*)) = p-temp-stk (*p*))
∧ (p-prog-segment (p-set-pc (*p*, *pc*)) = p-prog-segment (*p*))
∧ (p-data-segment (p-set-pc (*p*, *pc*)) = p-data-segment (*p*))
∧ (p-max-ctrl-stk-size (p-set-pc (*p*, *pc*)) = p-max-ctrl-stk-size (*p*))
∧ (p-max-temp-stk-size (p-set-pc (*p*, *pc*)) = p-max-temp-stk-size (*p*))
∧ (p-word-size (p-set-pc (*p*, *pc*)) = p-word-size (*p*))
∧ (p-psw (p-set-pc (*p*, *pc*)) = p-psw (*p*))

EVENT: Disable p-set-pc.

THEOREM: p-psw-not-run
(p-psw (*p-state*) ≠ 'run) → (p (*p-state*, *clock*) = *p-state*)

THEOREM: p-psw-p-halt-x-y-error-msg
p (p-halt (*p-state*, x-y-error-msg (*x*, *y*)), *n*)
= p-halt (*p-state*, x-y-error-msg (*x*, *y*))

EVENT: Disable p-psw-p-halt-x-y-error-msg.

THEOREM: p-accessors-p-run-subr
(p-prog-segment (p-run-subr (*subr*, *p*)) = p-prog-segment (*p*))
∧ (p-max-ctrl-stk-size (p-run-subr (*subr*, *p*)) = p-max-ctrl-stk-size (*p*))
∧ (p-max-temp-stk-size (p-run-subr (*subr*, *p*)) = p-max-temp-stk-size (*p*))
∧ (p-word-size (p-run-subr (*subr*, *p*)) = p-word-size (*p*))

THEOREM: p-accessors-lr-apply-subr
(p-pc (lr-apply-subr (*l1*, *l2*)) = p-pc (*l2*))
∧ (p-prog-segment (lr-apply-subr (*l1*, *l2*)) = p-prog-segment (*l2*))
∧ (p-max-ctrl-stk-size (lr-apply-subr (*l1*, *l2*))
= p-max-ctrl-stk-size (*l2*))
∧ (p-max-temp-stk-size (lr-apply-subr (*l1*, *l2*))
= p-max-temp-stk-size (*l2*))
∧ (p-word-size (lr-apply-subr (*l1*, *l2*)) = p-word-size (*l2*))

THEOREM: p-prog-segment-lr-eval
p-prog-segment (lr-eval (*flag*, *l*, *c*)) = p-prog-segment (*l*)

THEOREM: p-max-ctrl-stk-size-lr-eval
p-max-ctrl-stk-size (lr-eval (*flag*, *l*, *c*)) = p-max-ctrl-stk-size (*l*)

THEOREM: p-max-temp-stk-size-lr-eval
p-max-temp-stk-size (lr-eval (*flag*, *l*, *c*)) = p-max-temp-stk-size (*l*)

THEOREM: p-word-size-lr-eval
p-word-size (lr-eval (*flag*, *l*, *c*)) = p-word-size (*l*)

THEOREM: area-name-p-pc-lr-eval
area-name (p-pc (lr-eval (*flag*, *l*, *c*))) = area-name (p-pc (*l*))

THEOREM: lr-programs-properp-lr-eval
lr-programs-properp (lr-eval (*flag*, *l*, *c*), *table*)
= lr-programs-properp (*l*, *table*)

THEOREM: definedp-deposit
definedp (*tag*, deposit (*anything*, *addr*, *data-seg*)) = definedp (*tag*, *data-seg*)

THEOREM: deposit-a-list-cons-opener
deposit-a-list (cons (*x*, *list*), *addr*, *data-seg*)
= deposit (*x*, *addr*, deposit-a-list (*list*, add1-addr (*addr*), *data-seg*))

THEOREM: deposit-a-list-nil
deposit-a-list (**nil**, *addr*, *data-seg*) = *data-seg*

EVENT: Disable deposit-a-list.

THEOREM: assoc-put-assoc-3
assoc (*name1*, put-assoc (*val*, *name2*, *alist*))
= **if** *name1* = *name2*
then if definedp (*name1*, *alist*) **then** cons (*name1*, *val*)
else f endif
else assoc (*name1*, *alist*) **endif**

EVENT: Disable assoc-put-assoc-3.

THEOREM: adpp-lessp-offset-deposit
((*offset* < length (cdr (assoc (*name*, *data-seg*)))) ∧ definedp (*name*, *data-seg*))
→ (*offset* < length (cdr (assoc (*name*, deposit (*anything*, *anywhere*, *data-seg*)))))

THEOREM: adpp-deposit-anything-at-all
adpp (*adp*, *data-seg*) → adpp (*adp*, deposit (*anything*, *addr2*, *data-seg*))

EVENT: Disable adpp-lessp-offset-deposit.

EVENT: Disable adpp-deposit-anything-at-all.

THEOREM: adpp-untag-definedp-area-name
adpp (untag (*addr*), *data-seg*) → definedp (area-name (*addr*), *data-seg*)

EVENT: Disable adpp-untag-definedp-area-name.

THEOREM: adpp-cons-pack-definedp-area-name
adpp (cons (pack (*xxx*), *offset*), *data-seg*) → definedp (pack (*xxx*), *data-seg*)

THEOREM: adpp-untag-numberp-offset
adpp (untag (*addr*), *data-seg*) → (offset (*addr*) ∈ **N**)

EVENT: Disable adpp-untag-numberp-offset.

THEOREM: adpp-untag-listp
adpp (untag (*addr*), *data-seg*) → listp (untag (*addr*))

EVENT: Disable adpp-untag-listp.

THEOREM: adpp-add-addr-0
(adpp (untag (*addr*), *data-seg*)
∧ (cddr (*addr*) = **nil**)
∧ (type (*addr*) = 'addr)
∧ (*n* ≃ 0))
→ (add-addr (*addr*, *n*) = *addr*)

EVENT: Disable adpp-add-addr-0.

THEOREM: adpp-untag-lessp-offset
adpp (untag (*addr*), *data-seg*)
→ (offset (*addr*) < length (cdr (assoc (area-name (*addr*), *data-seg*))))

EVENT: Disable adpp-untag-lessp-offset.

THEOREM: adpp-same-signature
same-signature (*data-seg2*, *data-seg1*)
→ (adpp (*adp*, *data-seg2*) = adpp (*adp*, *data-seg1*))

EVENT: Disable adpp.

THEOREM: p-objectp-similar-p-states
(p-objectp (*object*, *p0*)
∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))
∧ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
∧ (p-word-size (*p0*) = p-word-size (*p1*)))
→ p-objectp (*object*, *p1*)

THEOREM: all-p-objectps-lr->p-similar-states
(all-p-objectps (*lst*, *p0*)
 ∧ (p-word-size (*p0*) = p-word-size (*p1*))
 ∧ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
 ∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*)))
→ all-p-objectps (*lst*, *p1*)

THEOREM: proper-p-data-segmentp-lr->p-similar-states
(proper-p-data-segmentp (*data-seg*, *p0*)
 ∧ (p-word-size (*p0*) = p-word-size (*p1*))
 ∧ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
 ∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*)))
→ proper-p-data-segmentp (*data-seg*, *p1*)

THEOREM: proper-p-temp-var-dclsp-lr->p-similar-states
(proper-p-temp-var-dclsp (*temp-var-dcls*, *p0*)
 ∧ (p-word-size (*p0*) = p-word-size (*p1*))
 ∧ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
 ∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*)))
→ proper-p-temp-var-dclsp (*temp-var-dcls*, *p1*)

THEOREM: proper-p-instructionp-similar-p-states
(proper-p-instructionp (*ins*, *name*, *p0*)
 ∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))
 ∧ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
 ∧ (p-word-size (*p0*) = p-word-size (*p1*)))
→ proper-p-instructionp (*ins*, *name*, *p1*)

THEOREM: proper-labeled-p-instructionsp-lr->p-similar-states
(proper-labeled-p-instructionsp (*lst*, *name*, *p0*)
 ∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))
 ∧ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
 ∧ (p-word-size (*p0*) = p-word-size (*p1*)))
→ proper-labeled-p-instructionsp (*lst*, *name*, *p1*)

THEOREM: proper-p-prog-segmentp-lr->p-similar-states
(proper-p-prog-segmentp (*programs*, *p0*)
 ∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))
 ∧ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
 ∧ (p-word-size (*p0*) = p-word-size (*p1*)))
→ proper-p-prog-segmentp (*programs*, *p1*)

THEOREM: proper-p-temp-stkp-lr->p-similar-states
(proper-p-temp-stkp (*temp-stk*, *p0*)
 ∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))

$\wedge$ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
$\wedge$ (p-word-size (*p0*) = p-word-size (*p1*)))
$\rightarrow$ proper-p-temp-stkp (*temp-stk*, *p1*)

THEOREM: proper-p-alistp-lr->p-similar-states
(proper-p-alistp (*bindings*, *p0*)
$\wedge$ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))
$\wedge$ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
$\wedge$ (p-word-size (*p0*) = p-word-size (*p1*)))
$\rightarrow$ proper-p-alistp (*bindings*, *p1*)

THEOREM: proper-p-ctrl-stkp-lr->p-similar-states
(proper-p-ctrl-stkp (*ctrl-stk*, *name*, *p0*)
$\wedge$ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))
$\wedge$ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
$\wedge$ (p-word-size (*p0*) = p-word-size (*p1*)))
$\rightarrow$ proper-p-ctrl-stkp (*ctrl-stk*, *name*, *p1*)

```
;; Now we prove what the result of running the SUBRPs.  We
;; start with a sample state (that the rewriter can match with
;; P-APPLY-SUBR-STATE) and run it.  We are only interested in TEMP-STK and
;; DATA-SEGMENT of the result.  However the running of the Piton code can
;; be a bit tedious, so we try and prove both parts at once with the
;; following function P-GOOD-RESULTP.  This also has the not ERRORP check
;; inside of it so that we should only have one instance of the Piton
;; interpreter (P) in each theorem.  This should hopefully reduce the time
;; (and pain) of proving these theorems.
```

DEFINITION:
p-good-resultp (*p*, *data-seg*, *temp-stk*, *ctrl-stk*, *pc*)
= **if** p-psw (*p*) $\neq$ 'run **then t**
else (p-data-segment (*p*) = *data-seg*)
$\wedge$ (p-temp-stk (*p*) = *temp-stk*)
$\wedge$ listp (*ctrl-stk*)
$\wedge$ (p-ctrl-stk (*p*) = *ctrl-stk*)
$\wedge$ (p-pc (*p*) = *pc*) **endif**

THEOREM: assoc-append-1
assoc (*x*, append (*list1*, *list2*))
= **if** definedp (*x*, *list1*) **then** assoc (*x*, *list1*)
else assoc (*x*, *list2*) **endif**

EVENT: Disable assoc-append-1.

THEOREM: lr-programs-properp-1-all-user-fnamesp-not-user-fnamep

(all-user-fnamesp (strip-cars (*programs*)) ∧ (¬ user-fnamep (*x*)))
→ (¬ definedp (*x*, *programs*))

THEOREM: definitions-subrps-lr-programs-properp
lr-programs-properp (*l*, *table*)
→ ((assoc ('car, comp-programs (p-prog-segment (*l*))) = P-CAR-CODE)
∧ (assoc ('cdr, comp-programs (p-prog-segment (*l*)))
= P-CDR-CODE)
∧ (assoc ('cons, comp-programs (p-prog-segment (*l*)))
= P-CONS-CODE)
∧ (assoc ('false, comp-programs (p-prog-segment (*l*)))
= P-FALSE-CODE)
∧ (assoc ('falsep, comp-programs (p-prog-segment (*l*)))
= P-FALSEP-CODE)
∧ (assoc ('listp, comp-programs (p-prog-segment (*l*)))
= P-LISTP-CODE)
∧ (assoc ('nlistp, comp-programs (p-prog-segment (*l*)))
= P-NLISTP-CODE)
∧ (assoc ('true, comp-programs (p-prog-segment (*l*)))
= P-TRUE-CODE)
∧ (assoc ('truep, comp-programs (p-prog-segment (*l*)))
= P-TRUEP-CODE))

EVENT: Disable lr-programs-properp-1-all-user-fnamesp-not-user-fnamep.

EVENT: Disable definitions-subrps-lr-programs-properp.

```
;; and now some openers for p-good-resultp
```

THEOREM: p-good-resultp-p-state-opener
p-good-resultp (p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
'run),
result-data-seg,
result-temp-stk,
result-ctrl-stk,
result-pc)
= ((*data-seg* = *result-data-seg*)

$\wedge$ (*temp-stk* = *result-temp-stk*)
$\wedge$ listp (*result-ctrl-stk*)
$\wedge$ (*ctrl-stk* = *result-ctrl-stk*)
$\wedge$ (*pc* = *result-pc*))

THEOREM: p-good-resultp-p-halt-errorp-opener
(*psw* $\neq$ 'run)
$\rightarrow$ p-good-resultp (p-halt (*p*, *psw*), *data-seg*, *temp-stk*, *ctrl-stk*, *pc*)

EVENT: Disable p-good-resultp.

THEOREM: all-p-objectps-bad-type
((get (*offset*, *lst*) $\neq$ list (type (get (*offset*, *lst*)), untag (get (*offset*, *lst*))))
$\wedge$ (*offset* $\in$ **N**)
$\wedge$ (*offset* $<$ length (*lst*)))
$\rightarrow$ ($\neg$ all-p-objectps (*lst*, *p*))

THEOREM: proper-p-data-segmentp-bad-type
((fetch (*addr*, *data-seg*)
$\neq$ list (type (fetch (*addr*, *data-seg*)), untag (fetch (*addr*, *data-seg*))))
$\wedge$ adpp (untag (*addr*), *data-seg*))
$\rightarrow$ ($\neg$ proper-p-data-segmentp (*data-seg*, *p*))

THEOREM: p-current-program-p-state
p-current-program (p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))
= assoc (area-name (*pc*), *prog-seg*)

THEOREM: p-current-instruction-opener
p-current-instruction (p-state (*pc*,
temp-stk,
ctrl-stk,
prog-segment,
data-segment,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,

psw))
= unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*), *prog-segment*))))

EVENT: Disable p-current-instruction-opener.

THEOREM: fetch-deposit
((offset (*addr1*) ∈ **N**) ∧ (offset (*addr2*) ∈ **N**))
→ (fetch (*addr1*, deposit (*value*, *addr2*, *data-seg*))
= **if** definedp (area-name (*addr2*), *data-seg*)
then if area-name (*addr1*) = area-name (*addr2*)
then if offset (*addr1*) = offset (*addr2*) **then** *value*
else fetch (*addr1*, *data-seg*) **endif**
else fetch (*addr1*, *data-seg*) **endif**
else fetch (*addr1*, *data-seg*) **endif**)

`;; add-addr`

THEOREM: area-name-add-addr
area-name (add-addr (*addr*, *n*)) = area-name (*addr*)

THEOREM: offset-add-addr
offset (add-addr (*addr*, *n*)) = (offset (*addr*) + *n*)

THEOREM: adp-name-untag-add-addr
adp-name (untag (add-addr (*addr*, *n*))) = area-name (*addr*)

THEOREM: add-addr-of-non-number
(*n* ∉ **N**) → (add-addr (*addr*, *n*) = add-addr (*addr*, 0))

THEOREM: add-addr-add-addr
add-addr (add-addr (*addr*, *n*), *m*) = add-addr (*addr*, *n* + *m*)

THEOREM: listp-untag-add-addr
listp (untag (add-addr (*addr*, *n*)))

THEOREM: type-add-addr
type (add-addr (*addr*, *n*)) = type (*addr*)

THEOREM: cddr-add-addr
cddr (add-addr (*addr*, *n*)) = **nil**

THEOREM: area-name-lr-return-pc
area-name (lr-return-pc (*l*)) = area-name (p-pc (*l*))

THEOREM: listp-untag-lr-return-pc
listp (untag (lr-return-pc (*l*)))

THEOREM: type-lr-return-pc
type (lr-return-pc (*l*)) = 'pc

THEOREM: cddr-lr-return-pc
cddr (lr-return-pc (*l*)) = **nil**

THEOREM: numberp-offset-return-pc
offset (lr-return-pc (*l*)) ∈ **N**

THEOREM: numberp-cdr-untag-return-pc
cdr (untag (lr-return-pc (*l*))) ∈ **N**

THEOREM: car-untag-lr-return-pc
car (untag (lr-return-pc (*l*))) = car (untag (p-pc (*l*)))

;; sub-addr

THEOREM: area-name-sub-addr
area-name (sub-addr (*addr*, *n*)) = area-name (*addr*)

THEOREM: cddr-sub-addr
cddr (sub-addr (*addr*, *n*)) = **nil**

THEOREM: type-sub-addr
type (sub-addr (*addr*, *n*)) = type (*addr*)

THEOREM: listp-untag-sub-addr
listp (untag (sub-addr (*addr*, *n*)))

THEOREM: offset-sub-addr
offset (sub-addr (*addr*, *n*)) = (offset (*addr*) − *n*)

;; LR-BOUNDARY-NODEP

THEOREM: lr-boundary-nodep-sub-addr
lr-boundary-nodep (*addr*)
→ lr-boundary-nodep (sub-addr (*addr*, identity (LR-NODE-SIZE)))

THEOREM: lr-boundary-nodep-add-addr-lr-node-size
lr-boundary-nodep (*addr*)
→ lr-boundary-nodep (add-addr (*addr*, identity (LR-NODE-SIZE)))

EVENT: Disable lr-boundary-nodep.

;; LR-NODEP

THEOREM: lr-nodep-opener
lr-nodep (*addr*, *data-seg*)
= ((type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ adpp (untag (*addr*), *data-seg*)
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = identity (LR-HEAP-NAME)))

EVENT: Disable lr-nodep.

;; LR-GOOD-POINTERP

THEOREM: lr-good-pointerp-opener
lr-good-pointerp (*addr*, *data-seg*)
= ((type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ adpp (untag (*addr*), *data-seg*)
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = identity (LR-HEAP-NAME))
∧ (type (fetch (add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)),
data-seg))
= 'nat))

EVENT: Disable lr-good-pointerp.

THEOREM: equal-plus-remainder-0-fact
(((*offset1* **mod** *max*) = 0)
∧ ((*offset2* **mod** *max*) = 0)
∧ (*n* < *max*)
∧ (*m* < *max*)
∧ (*offset1* ∈ **N**)
∧ (*offset2* ∈ **N**))
→ (((*n* + *offset1*) = (*m* + *offset2*))
= ((fix (*n*) = fix (*m*)) ∧ (*offset1* = *offset2*))))

THEOREM: lr-boundary-offsetp-equal-plus-fact
(lr-boundary-offsetp (*offset1*)
∧ lr-boundary-offsetp (*offset2*)
∧ (*n* < LR-NODE-SIZE)
∧ (*m* < LR-NODE-SIZE)

$\wedge$ (*offset1* $\in$ **N**)
$\wedge$ (*offset2* $\in$ **N**))
$\rightarrow$ (((*n* + *offset1*) = (*m* + *offset2*))
= ((fix (*n*) = fix (*m*)) $\wedge$ (*offset1* = *offset2*)))

THEOREM: good-posp-list-nx-t-simple
(good-posp (`'list`, *pos*, *body*)
$\wedge$ listp (*pos*)
$\wedge$ (car (last (*pos*)) < length (cur-expr (butlast (*pos*), *body*))))
$\rightarrow$ (good-posp (`'list`, nx (*pos*), *body*) $\wedge$ good-posp1 (*pos*, *body*))

THEOREM: lr-programs-properp-1-lr-proper-exprp
(lr-programs-properp-1 (*progs*, *program-names*, *table*) $\wedge$ (*prog* $\in$ *progs*))
$\rightarrow$ lr-proper-exprp (**t**,
program-body (*prog*),
program-names,
formal-vars (*prog*),
strip-cars (temp-var-dcls (*prog*)),
table)

THEOREM: lr-proper-exprp-list-lr-proper-get-t
lr-proper-exprp (`'list`, *expr*, *pnames*, *formals*, *temps*, *table*)
$\rightarrow$ (lr-proper-exprp (**t**, get (*n*, *expr*), *pnames*, *formals*, *temps*, *table*)
= (*n* < length (*expr*)))

THEOREM: lr-proper-exprp-t-lr-proper-get-t
((car (*expr*) $\neq$ S-TEMP-FETCH)
$\wedge$ (car (*expr*) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (*expr*) $\neq$ S-TEMP-TEST)
$\wedge$ (car (*expr*) $\neq$ `'quote`)
$\wedge$ (*n* $\not\simeq$ `0`)
$\wedge$ lr-proper-exprp (**t**, *expr*, *pnames*, *formals*, *temps*, *table*))
$\rightarrow$ (lr-proper-exprp (**t**, get (*n*, *expr*), *pnames*, *formals*, *temps*, *table*)
= (*n* < length (*expr*)))

EVENT: Disable lr-proper-exprp-list-lr-proper-get-t.

THEOREM: lr-proper-exprp-lr-proper-exprp-cur-expr
(lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*) $\wedge$ good-posp1 (*pos*, *body*))
$\rightarrow$ lr-proper-exprp (**t**, cur-expr (*pos*, *body*), *pnames*, *formals*, *temps*, *table*)

THEOREM: lr-programs-properp-lr-programs-properp-1
(lr-programs-properp (*l*, *table*) $\wedge$ (*prog-seg* = p-prog-segment (*l*)))
$\rightarrow$ (lr-programs-properp-1 (p-prog-segment (*l*),
strip-logic-fnames (cdr (*prog-seg*)),

table)
∧ definedp (area-name (p-pc (*l*)), *prog-seg*))

THEOREM: lr-programs-properp-lr-proper-exprp-lr-expr
(lr-programs-properp (*l*, *table*)
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*))))
→ lr-proper-exprp (**t**,
lr-expr (*l*),
strip-logic-fnames (cdr (p-prog-segment (*l*))),
formal-vars (p-current-program (*l*)),
strip-cars (temp-var-dcls (p-current-program (*l*))),
table)

THEOREM: lr-proper-exprp-length-cur-expr
(lr-proper-exprp (**t**, *expr*, *pnames*, *formals*, *temps*, *table*)
∧ listp (*expr*)
∧ (subrp (car (*expr*)) ∨ body (car (*expr*)))
∧ (car (*expr*) ≠ `'quote`))
→ (length (*expr*) = (1 + arity (car (*expr*))))

THEOREM: listp-comp-body-1
listp (comp-body-1 (*flag*, *body*, *n*))
= **if** *flag* = `'list` **then** listp (*body*)
else t endif

THEOREM: car-append
listp (*x*) → (car (append (*x*, *y*)) = car (*x*))

THEOREM: length-cdr-comp-if-comp-body
length (comp-if (comp-body-1 (**t**, *test*, *n1*),
comp-body-1 (**t**, *then*, *n2*),
comp-body-1 (**t**, *else*, *n3*),
n))
= (length (comp-body-1 (**t**, *test*, *n1*))
+ length (comp-body-1 (**t**, *then*, *n2*))
+ length (comp-body-1 (**t**, *else*, *n3*))
+ `4`)

THEOREM: lr-p-c-size-list-0-opener
lr-p-c-size-list (`0`, *expr*) = `0`

THEOREM: lr-p-c-size-list-add1-opener
((1 + *n*) < length (*expr*))
→ (lr-p-c-size-list (1 + *n*, *expr*)
= (lr-p-c-size (**t**, cadr (*expr*)) + lr-p-c-size-list (*n*, cdr (*expr*))))

THEOREM: length-comp-body-1-lr-p-c-size
length (comp-body-1 (*flag*, *body*, *n*)) = lr-p-c-size (*flag*, *body*)

EVENT: Disable lr-p-c-size-list-add1-opener.

THEOREM: length-label-instrs
length (label-instrs (*instrs*, *n*)) = length (*instrs*)

THEOREM: length-comp-body-lr-p-c-size
length (comp-body (*body*)) = (1 + lr-p-c-size (**t**, *body*))

THEOREM: lr-p-c-size-flag-list
lr-p-c-size (`'list`, cdr (*expr*)) = lr-p-c-size-list (length (*expr*) − 1, *expr*)

THEOREM: lr-proper-exprp-car-if-cadr
(lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*)
∧ listp (*body*)
∧ (car (*body*) = `'if`))
→ lr-proper-exprp (**t**, cadr (*body*), *pnames*, *formals*, *temps*, *table*)

THEOREM: lr-proper-exprp-car-if-caddr
(lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*)
∧ listp (*body*)
∧ (car (*body*) = `'if`))
→ lr-proper-exprp (**t**, caddr (*body*), *pnames*, *formals*, *temps*, *table*)

THEOREM: lr-proper-exprp-car-if-cadddr
(lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*)
∧ listp (*body*)
∧ (car (*body*) = `'if`))
→ lr-proper-exprp (**t**, cadddr (*body*), *pnames*, *formals*, *temps*, *table*)

THEOREM: good-posp-list-t-offset-program-body
(good-posp (`'list`, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ listp (lr-expr-list (*l*))
∧ listp (offset (p-pc (*l*))))
→ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))

THEOREM: good-posp-list-nx-offset-program-body
(good-posp (`'list`, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ listp (offset (p-pc (*l*)))
∧ listp (lr-expr-list (*l*)))
→ good-posp (`'list`,
nx (offset (p-pc (*l*))),
program-body (p-current-program (*l*)))

THEOREM: name-formal-vars-temp-var-dcls-program-body-cons
(name (cons (*name*, *rest*)) = *name*)
∧ (formal-vars (cons (*name*, cons (*formal-vars*, *rest*))) = *formal-vars*)
∧ (temp-var-dcls (cons (*name*,
cons (*formal-vars*, cons (*temp-var-dcls*, *program-body*))))
= *temp-var-dcls*)
∧ (program-body (cons (*name*,
cons (*formal-vars*, cons (*temp-var-dcls*, *program-body*))))
= *program-body*)

THEOREM: program-body-assoc-comp-programs-1
definedp (*name*, *programs*)
→ (program-body (assoc (*name*, comp-programs-1 (*programs*)))
= comp-body (program-body (assoc (*name*, *programs*))))

THEOREM: program-body-assoc-comp-programs
definedp (*name*, *programs*)
→ (program-body (assoc (*name*, comp-programs (*programs*)))
= **if** *name* = name (car (*programs*))
then label-instrs (append (comp-body-1 (**t**,
program-body (car (*programs*)),
0),
list (identity (list (`'set-global`,
area-name (LR-ANSWER-ADDR))),
`'(ret)`)),
0)
else comp-body (program-body (assoc (*name*, cdr (*programs*)))) **endif**)

THEOREM: definedp-area-name-member-p-current-program
definedp (area-name (p-pc (*l*)), p-prog-segment (*l*))
→ (p-current-program (*l*) ∈ p-prog-segment (*l*))

DEFINITION:
induct-hint-6 (*n*, *body*)
= **if** *n* < length (*body*) **then** induct-hint-6 (1 + *n*, *body*)
else t endif

THEOREM: lr-p-c-size-list-0
listp (*body*)
→ ((lr-p-c-size-list (*n*, *body*) = 0) = ((*n* ≃ 0) ∨ (cdr (*body*) ≃ **nil**)))

EVENT: Disable lr-p-c-size-list-0.

THEOREM: lessp-lr-p-c-size-list-lessp-sub1-length
(lr-p-c-size-list (*n*, *body*) ≠ 0)

$\rightarrow$ ((lr-p-c-size-list (n, *body*) $-$ 1)
$<$ lr-p-c-size-list (length (*body*) $-$ 1, *body*))

THEOREM: lr-p-pc-1-body-0
lr-p-pc-1 (0, *pos*) = 0

THEOREM: lessp-lr-p-pc-1-lr-p-c-size-helper-1
(listp (*body*)
$\wedge$ ($n \not\simeq$ 0)
$\wedge$ (lr-p-pc-1 (get (n, *body*), *pos*) $<$ lr-p-c-size (**t**, get (n, *body*)))
$\wedge$ (lr-p-pc-1 (get (n, *body*), *pos*) $\neq$ 0))
$\rightarrow$ (((lr-p-c-size-list ($n - 1$, *body*) + lr-p-pc-1 (get (n, *body*), *pos*)) $-$ 1)
$<$ lr-p-c-size-list (length (*body*) $-$ 1, *body*))

THEOREM: lessp-lr-p-pc-1-lr-p-c-size
lr-p-pc-1 (*body*, *pos*) $<$ lr-p-c-size (**t**, *body*)

EVENT: Disable lessp-lr-p-pc-1-lr-p-c-size-helper-1.

THEOREM: not-lessp-p-max-temp-stk-size-lr-push-tstk
(p-psw (lr-push-tstk (l, *anything*)) = 'run)
$\rightarrow$ (p-max-temp-stk-size (l)
$\not<$ length (p-temp-stk (lr-push-tstk (l, *anything*))))

THEOREM: proper-p-temp-stkp-lr->p-lr-push-tstk
proper-p-temp-stkp (*temp-stkp*, lr->p (lr-push-tstk (l, *anything*)))
= proper-p-temp-stkp (*temp-stkp*, lr->p (l))

THEOREM: proper-p-alistp-p-objectp
(proper-p-alistp (*bindings*, l) $\wedge$ definedp (*name*, *bindings*))
$\rightarrow$ p-objectp (cdr (assoc (*name*, *bindings*)), l)

THEOREM: formal-vars-assoc-comp-programs-1
definedp (*name*, *programs*)
$\rightarrow$ (formal-vars (assoc (*name*, comp-programs-1 (*programs*)))
= formal-vars (assoc (*name*, *programs*)))

THEOREM: formal-vars-assoc-comp-programs
definedp (*name*, *programs*)
$\rightarrow$ (formal-vars (assoc (*name*, comp-programs (*programs*)))
= formal-vars (assoc (*name*, *programs*)))

THEOREM: temp-var-dcls-assoc-comp-programs-1
definedp (*name*, *programs*)
$\rightarrow$ (temp-var-dcls (assoc (*name*, comp-programs-1 (*programs*)))
= temp-var-dcls (assoc (*name*, *programs*)))

THEOREM: temp-var-dcls-assoc-comp-programs
definedp (*name*, *programs*)
→ (temp-var-dcls (assoc (*name*, comp-programs (*programs*)))
= temp-var-dcls (assoc (*name*, *programs*)))

THEOREM: lr-programs-properp-definedp-car-untag-p-pc
lr-programs-properp (*l*, *table*)
→ definedp (car (untag (p-pc (*l*))), p-prog-segment (*l*))

THEOREM: p-objectp-cdr-assoc-litatom-proper-p-alistp
(proper-p-alistp (*bindings*, *lp*)
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (strip-cars (*bindings*)
= append (formal-vars (assoc (car (untag (p-pc (*l*))),
comp-programs-1 (p-prog-segment (*l*)))),
strip-cars (temp-var-dcls (assoc (car (untag (p-pc (*l*))),
comp-programs-1 (p-prog-segment (*l*)))))))
∧ litatom (lr-expr (*l*)))
→ p-objectp (cdr (assoc (lr-expr (*l*), *bindings*)), *lp*)

THEOREM: proper-p-temp-stkp-lr-push-tstk-assoc-bindings
(lr-programs-properp (*l*, *table*)
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ proper-p-alistp (bindings (car (p-ctrl-stk (*l*))), lr->p (*l*))
∧ litatom (lr-expr (*l*))
∧ (strip-cars (bindings (car (p-ctrl-stk (*l*))))
= append (formal-vars (assoc (car (untag (p-pc (*l*))),
comp-programs-1 (p-prog-segment (*l*)))),
strip-cars (temp-var-dcls (assoc (car (untag (p-pc (*l*))),
comp-programs-1 (p-prog-segment (*l*))))))))
→ (proper-p-temp-stkp (p-temp-stk (lr-push-tstk (*l*,
cdr (assoc (lr-expr (*l*),
bindings (car (p-ctrl-stk (*l*)))))),
lr->p (*l*))
= proper-p-temp-stkp (p-temp-stk (*l*), lr->p (*l*)))

THEOREM: lr-p-pc-lr-push-tstk
lr-p-pc (lr-push-tstk (*l*, *anything*)) = lr-p-pc (*l*)

THEOREM: proper-p-statep-lr->p-lr-push-tstk
(proper-p-statep (lr->p (*l*))
∧ lr-programs-properp (*l*, *table*)
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ (p-psw (lr-push-tstk (*l*,

cdr (assoc (lr-expr (*l*),
bindings (car (p-ctrl-stk (*l*)))))))
= 'run)
∧ litatom (lr-expr (*l*)))
→ proper-p-statep (lr->p (lr-push-tstk (*l*,
cdr (assoc (lr-expr (*l*),
bindings (car (p-ctrl-stk (*l*))))))))

THEOREM: good-posp1-cons-lessp-4-if-lr-proper-exprp
((car (cur-expr (*pos*, *body*)) = 'if)
∧ good-posp1 (*pos*, *body*)
∧ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*))
→ (good-posp1 (dv (*pos*, 1), *body*)
∧ good-posp1 (dv (*pos*, 2), *body*)
∧ good-posp1 (dv (*pos*, 3), *body*))

THEOREM: good-posp-cons-lessp-4-if-lr-programs-properp
((car (lr-expr (*l*)) = 'if)
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*))
→ (good-posp1 (dv (offset (p-pc (*l*)), 1), program-body (p-current-program (*l*)))
∧ good-posp1 (dv (offset (p-pc (*l*)), 2),
program-body (p-current-program (*l*)))
∧ good-posp1 (dv (offset (p-pc (*l*)), 3),
program-body (p-current-program (*l*))))

THEOREM: proper-p-statep-lr->p-lr-set-pos
(lr-programs-properp (*l*, *table*) ∧ proper-p-statep (lr->p (*l*)))
→ proper-p-statep (lr->p (lr-set-pos (*l*, *pos*)))

THEOREM: lr-p-pc-lr-pop-tstk
lr-p-pc (lr-pop-tstk (*l*)) = lr-p-pc (*l*)

THEOREM: proper-p-statep-lr->p-lr-pop-tstk
proper-p-statep (lr->p (*l*)) → proper-p-statep (lr->p (lr-pop-tstk (*l*)))

THEOREM: good-posp-dv-1-temps-lr-expr
(((car (lr-expr (*l*)) = S-TEMP-EVAL) ∨ (car (lr-expr (*l*)) = S-TEMP-TEST))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*))))
→ good-posp1 (dv (offset (p-pc (*l*)), 1), program-body (p-current-program (*l*)))

THEOREM: proper-p-alistp-put-assoc
(proper-p-alistp (*bindings*, *l*) ∧ p-objectp (*object*, *l*))
→ proper-p-alistp (put-assoc (*object*, *var-name*, *bindings*), *l*)

THEOREM: listp-p-temp-stk-lr-push-tstk
(p-psw (lr-push-tstk (l, *object*)) = 'run)
→ listp (p-temp-stk (lr-push-tstk (l, *object*)))

THEOREM: lr-p-pc-lr-set-temp
lr-p-pc (lr-set-temp (l, *value*, *var-name*)) = lr-p-pc (l)

THEOREM: proper-p-statep-lr-set-temp
(proper-p-statep (lr->p (l)) ∧ listp (p-temp-stk (l)))
→ proper-p-statep (lr->p (lr-set-temp (l, car (p-temp-stk (l)), *var-name*)))

THEOREM: p-objectp-cdr-assoc-bindings-proper-p-alistp
(proper-p-alistp (*bindings*, l) ∧ definedp (*object*, *bindings*))
→ p-objectp (cdr (assoc (*object*, *bindings*)), l)

THEOREM: definedp-caddr-lr-expr-bindings-ctrl-stk
(lr-programs-properp-1 (*progs*, *program-names*, *table*)
∧ definedp (*name*, *progs*)
∧ ((car (cur-expr (*pos*, program-body (assoc (*name*, *progs*))))
= S-TEMP-FETCH)
∨ (car (cur-expr (*pos*, program-body (assoc (*name*, *progs*))))
= S-TEMP-TEST))
∧ good-posp1 (*pos*, program-body (assoc (*name*, *progs*)))
∧ (strip-cars (*bindings*)
= append (formal-vars (assoc (*name*, comp-programs (*progs*))),
strip-cars (temp-var-dcls (assoc (*name*,
comp-programs (*progs*)))))))
→ definedp (caddr (cur-expr (*pos*, program-body (assoc (*name*, *progs*)))),
bindings)

THEOREM: proper-p-temp-stkp-p-temp-stk-lr-do-temp-fetch
(proper-p-framep (top (p-ctrl-stk (*l1*)), area-name (p-pc (*l1*)), *l2*)
∧ lr-programs-properp (*l1*, *table*)
∧ ((car (lr-expr (*l1*)) = S-TEMP-FETCH)
∨ (car (lr-expr (*l1*)) = S-TEMP-TEST))
∧ good-posp1 (offset (p-pc (*l1*)), program-body (p-current-program (*l1*)))
∧ same-signature (p-data-segment (*l1*), p-data-segment (*l2*))
∧ (p-prog-segment (lr->p (*l1*)) = p-prog-segment (*l2*))
∧ (p-word-size (*l1*) = p-word-size (*l2*)))
→ (proper-p-temp-stkp (p-temp-stk (lr-do-temp-fetch (*l1*)), *l2*)
= proper-p-temp-stkp (p-temp-stk (*l1*), *l2*))

THEOREM: length-lr-do-temp-fetch
(p-psw (lr-do-temp-fetch (l)) = 'run)
→ (p-max-temp-stk-size (l) ≮ length (p-temp-stk (lr-do-temp-fetch (l))))

THEOREM: lr-p-pc-lr-do-temp-fetch
lr-p-pc (lr-do-temp-fetch (*l*)) = lr-p-pc (*l*)

THEOREM: proper-p-statep-lr-do-temp-fetch
((p-psw (lr-do-temp-fetch (*l*)) = 'run)
 ∧ lr-programs-properp (*l*, *table*)
 ∧ ((car (lr-expr (*l*)) = S-TEMP-FETCH)
  ∨ (car (lr-expr (*l*)) = S-TEMP-TEST))
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ proper-p-statep (lr->p (*l*)))
→ proper-p-statep (lr->p (lr-do-temp-fetch (*l*)))

THEOREM: length-lr-push-tstk
(p-psw (lr-push-tstk (*l*, *object*)) = 'run)
→ (p-max-temp-stk-size (*l*)
  ≮ length (p-temp-stk (lr-push-tstk (*l*, *object*))))

THEOREM: listp-p-temp-stk-lr-do-temp-fetch
(p-psw (lr-do-temp-fetch (*l*)) = 'run)
→ listp (p-temp-stk (lr-do-temp-fetch (*l*)))

THEOREM: proper-p-prog-segmentp-append
plistp (*segment1*)
→ (proper-p-prog-segmentp (append (*segment1*, *segment2*), *p*)
  = (proper-p-prog-segmentp (*segment1*, *p*)
   ∧ proper-p-prog-segmentp (*segment2*, *p*)))

THEOREM: lr-programs-properp-expr-quote-type-addr
(lr-programs-properp (*l*, *table*)
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ listp (lr-expr (*l*))
 ∧ (car (lr-expr (*l*)) = 'quote))
→ (type (cadr (lr-expr (*l*))) = 'addr)

THEOREM: proper-p-instructionp-push-constant-opener
proper-p-instructionp (list ('push-constant, *object*), *name*, *p*)
= proper-p-push-constant-instructionp (list ('push-constant, *object*),
                                       name,
                                       p)

THEOREM: proper-labeled-p-instructionsp-find-labelp-non-litatom
(proper-labeled-p-instructionsp (*body*, *name*, *p*) ∧ (¬ litatom (*label*)))
→ (find-labelp (*label*, *body*) = **f**)

THEOREM: lessp-4-not-zerop-not-1-not-2-3
((*n* ≮ 0) ∧ (*n* ≠ 1) ∧ (*n* ≠ 2) ∧ (*n* < 4)) → (*n* = 3)

THEOREM: lessp-4-not-zerop-not-1-not-2-3-get-car-pos
((car (*pos*) $\not\simeq$ 0)
 $\wedge$ (car (*pos*) $\neq$ 1)
 $\wedge$ (car (*pos*) $\neq$ 2)
 $\wedge$ (car (*pos*) $<$ 4))
$\rightarrow$ (get (car (*pos*), *body*) = cadddr (*body*))

EVENT: Disable lessp-4-not-zerop-not-1-not-2-3-get-car-pos.

THEOREM: lessp-index-lessp-lr-p-c-size-list
lr-p-c-size-list (length (cdr (*body*)), *body*) $\not<$ lr-p-c-size-list (*n*, *body*)

THEOREM: lessp-plus-lr-p-c-size-lr-p-pc-1-helper
(listp (*body*)
 $\wedge$ (*n* $\not\simeq$ 0)
 $\wedge$ (lr-p-c-size (**t**, get (*n*, *body*)) $\not<$ *x*)
 $\wedge$ ((*n* $-$ 1) $<$ length (cdr (*body*)))
 $\wedge$ (*len* = length (cdr (*body*))))
$\rightarrow$ (((lr-p-c-size-list (*len*, *body*) + 1)
 $<$ (lr-p-c-size-list (*n* $-$ 1, *body*) + *x*))
 = **f**)

THEOREM: lessp-plus-lr-p-c-size-lr-p-pc-1
(good-posp1 (*pos*, *body*) $\wedge$ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*))
$\rightarrow$ (lr-p-c-size (**t**, *body*)
 $\not<$ (lr-p-pc-1 (*body*, *pos*) + lr-p-c-size (**t**, cur-expr (*pos*, *body*))))

DEFINITION:
induct-hint-7 (*pos*, *expr*, *n*)
= **if** *pos* $\simeq$ **nil then t**
 elseif *expr* $\simeq$ **nil then t**
 elseif car (*expr*) = 'if
 then let *then-n* **be** *n* + 3 + lr-p-c-size (**t**, cadr (*expr*))
 in
 case on car (*pos*):
 case = 1
 then induct-hint-7 (cdr (*pos*), cadr (*expr*), *n*)
 case = 2
 then induct-hint-7 (cdr (*pos*), caddr (*expr*), *then-n*)
 otherwise induct-hint-7 (cdr (*pos*),
 cadddr (*expr*),
 1
 + *then-n*
 + lr-p-c-size (**t**, caddr (*expr*))) **endcase endlet**

elseif car (*expr*) = S-TEMP-FETCH **then t**
elseif car (*expr*) = S-TEMP-EVAL
then induct-hint-7 (cdr (*pos*), cadr (*expr*), *n*)
elseif car (*expr*) = S-TEMP-TEST
then induct-hint-7 (cdr (*pos*), cadr (*expr*), *n* + 4)
elseif car (*expr*) = `'quote` **then t**
else induct-hint-7 (cdr (*pos*),
get (car (*pos*), *expr*),
n + lr-p-c-size-list (car (*pos*) − 1, *expr*)) **endif**

THEOREM: lr-p-c-size-s-temp-test-eval-cadr-not-lessp-fact
(listp (*expr*)
∧ ((car (*expr*) = S-TEMP-EVAL) ∨ (car (*expr*) = S-TEMP-TEST)))
→ (lr-p-c-size (**t**, cadr (*expr*)) < lr-p-c-size (**t**, *expr*))

THEOREM: length-comp-temp-test
(listp (*body*) ∧ (car (*body*) = S-TEMP-TEST))
→ (length (comp-temp-test (*any-body*, comp-body-1 (**t**, cadr (*body*), *n*), *any-n*))
= lr-p-c-size (**t**, *body*))

THEOREM: plistp-comp-temp-test
plistp (comp-temp-test (*body*, *instrs*, *n*))

THEOREM: length-comp-if-alt
(listp (*body*) ∧ (car (*body*) = `'if`))
→ (length (comp-if (comp-body-1 (**t**, cadr (*body*), *n1*),
comp-body-1 (**t**, caddr (*body*), *n2*),
comp-body-1 (**t**, cadddr (*body*), *n3*),
any-n))
= lr-p-c-size (**t**, *body*))

THEOREM: plistp-comp-if
(plistp (*else-instrs*) ∧ listp (*else-instrs*))
→ plistp (comp-if (*test-intrs*, *then-instrs*, *else-instrs*, *n*))

THEOREM: plistp-comp-body-1
plistp (comp-body-1 (*flag*, *body*, *n*))

THEOREM: lr-p-c-size-list-funcall-not-lessp-fact
(listp (*expr*)
∧ (car (*expr*) ≠ S-TEMP-FETCH)
∧ (car (*expr*) ≠ S-TEMP-EVAL)
∧ (car (*expr*) ≠ S-TEMP-TEST)
∧ (car (*expr*) ≠ `'quote`)
∧ (car (*expr*) ≠ `'if`))
→ (lr-p-c-size-list (length (*expr*) − 1, *expr*) < lr-p-c-size (**t**, *expr*))

THEOREM: lr-p-c-size-nlistp-body
$(\neg\ \text{listp}\,(\mathit{body})) \rightarrow (\text{lr-p-c-size}\,(\mathbf{t},\ \mathit{body}) = \texttt{1})$

THEOREM: firstn-lr-p-c-size-restn-lr-p-pc-1-comp-body-1-helper-1
$$\begin{array}{ll}
(\text{good-posp1}\,(\mathit{pos},\ \text{cadr}\,(\mathit{body})) & \\
\wedge & (\text{car}\,(\mathit{body}) = \texttt{'if}) \\
\wedge & \text{listp}\,(\mathit{body}) \\
\wedge & \text{lr-proper-exprp}\,(\mathbf{t},\ \mathit{body},\ \mathit{pnames},\ \mathit{formals},\ \mathit{temps},\ \mathit{table})) \\
\rightarrow & (\text{firstn}\,(\text{lr-p-c-size}\,(\mathbf{t},\ \text{cur-expr}\,(\mathit{pos},\ \text{cadr}\,(\mathit{body}))), \\
& \quad \text{restn}\,(\text{lr-p-pc-1}\,(\text{cadr}\,(\mathit{body}),\ \mathit{pos}), \\
& \quad\quad \text{comp-if}\,(\text{comp-body-1}\,(\mathbf{t},\ \text{cadr}\,(\mathit{body}),\ n), \\
& \quad\quad\quad \mathit{then\text{-}instrs}, \\
& \quad\quad\quad \mathit{else\text{-}instrs}, \\
& \quad\quad\quad n))) \\
= & \text{firstn}\,(\text{lr-p-c-size}\,(\mathbf{t},\ \text{cur-expr}\,(\mathit{pos},\ \text{cadr}\,(\mathit{body}))), \\
& \quad \text{restn}\,(\text{lr-p-pc-1}\,(\text{cadr}\,(\mathit{body}),\ \mathit{pos}), \\
& \quad\quad \text{comp-body-1}\,(\mathbf{t},\ \text{cadr}\,(\mathit{body}),\ n))))
\end{array}$$

THEOREM: firstn-restn-plus-comp-if-1
$$\begin{array}{ll}
((j = \text{length}\,(\mathit{test})) & \\
\wedge & \text{listp}\,(\mathit{then}) \\
\wedge & (m < \text{length}\,(\mathit{then})) \\
\wedge & (m \in \mathbf{N}) \\
\wedge & \text{listp}\,(\mathit{test}) \\
\wedge & (\text{length}\,(\mathit{then}) \not< (k + m))) \\
\rightarrow & (\text{firstn}\,(k,\ \text{restn}\,(\texttt{3} + j + m,\ \text{comp-if}\,(\mathit{test},\ \mathit{then},\ \mathit{else},\ n))) \\
= & \text{firstn}\,(k,\ \text{restn}\,(m,\ \mathit{then})))
\end{array}$$

THEOREM: firstn-unlabel-instrs-comp-body-1-lr-p-pc-1-helper-2
$$\begin{array}{ll}
(\text{good-posp1}\,(\mathit{pos},\ \text{caddr}\,(\mathit{body})) & \\
\wedge & (\text{car}\,(\mathit{body}) = \texttt{'if}) \\
\wedge & \text{listp}\,(\mathit{body}) \\
\wedge & \text{lr-proper-exprp}\,(\mathbf{t},\ \mathit{body},\ \mathit{pnames},\ \mathit{formals},\ \mathit{temps},\ \mathit{table})) \\
\rightarrow & (\text{lr-p-c-size}\,(\mathbf{t},\ \text{caddr}\,(\mathit{body})) \\
\not< & (\text{lr-p-pc-1}\,(\text{caddr}\,(\mathit{body}),\ \mathit{pos}) \\
+ & \text{lr-p-c-size}\,(\mathbf{t},\ \text{cur-expr}\,(\mathit{pos},\ \text{caddr}\,(\mathit{body})))))
\end{array}$$

THEOREM: firstn-restn-plus-comp-if-2
$$\begin{array}{ll}
((j = \text{length}\,(\mathit{test})) & \\
\wedge & (i = \text{length}\,(\mathit{then})) \\
\wedge & \text{listp}\,(\mathit{then}) \\
\wedge & (m < \text{length}\,(\mathit{else})) \\
\wedge & (m \in \mathbf{N}) \\
\wedge & \text{listp}\,(\mathit{test}) \\
\wedge & \text{listp}\,(\mathit{else})
\end{array}$$

$\wedge$ (length (*else*) $\not<$ (m + k)))
$\rightarrow$ (firstn (k, restn (j + i + m + 4, comp-if (*test*, *then*, *else*, n)))
= firstn (k, restn (m, *else*)))

THEOREM: firstn-unlabel-instrs-comp-body-1-lr-p-pc-1-helper-3
(good-posp1 (*pos*, cadddr (*body*))
$\wedge$ (car (*body*) = 'if)
$\wedge$ listp (*body*)
$\wedge$ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*))
$\rightarrow$ (lr-p-c-size (**t**, cadddr (*body*))
$\not<$ (lr-p-pc-1 (cadddr (*body*), *pos*)
+ lr-p-c-size (**t**, cur-expr (*pos*, cadddr (*body*)))))

THEOREM: plus-constant-fact-helper-1
(1 + n + 3 + x + y) = (n + 4 + x + y)

THEOREM: firstn-lr-p-c-size-restn-lr-p-pc-1-comp-body-1-helper-4
(good-posp1 (*pos*, cadr (*body*))
$\wedge$ listp (*body*)
$\wedge$ (car (*body*) = S-TEMP-TEST)
$\wedge$ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*))
$\rightarrow$ (firstn (lr-p-c-size (**t**, cur-expr (*pos*, cadr (*body*))),
restn (lr-p-pc-1 (cadr (*body*), *pos*) + 4,
comp-temp-test (*body-1*, comp-body-1 (**t**, cadr (*body*), n), m)))
= firstn (lr-p-c-size (**t**, cur-expr (*pos*, cadr (*body*))),
restn (lr-p-pc-1 (cadr (*body*), *pos*),
comp-body-1 (**t**, cadr (*body*), n))))

THEOREM: firstn-lr-p-c-size-restn-lr-p-pc-1-comp-body-1-helper-5
(4 + x) = (x + 4)

THEOREM: good-posp1-lr-proper-exprp-get-cadddr
(listp (*pos*)
$\wedge$ listp (*body*)
$\wedge$ (car (*body*) = 'if)
$\wedge$ (car (*pos*) $\neq$ 1)
$\wedge$ (car (*pos*) $\neq$ 2)
$\wedge$ (car (*pos*) $\neq$ 0)
$\wedge$ (car (*pos*) $\in$ **N**)
$\wedge$ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*)
$\wedge$ ((((car (*pos*) $-$ 1) $-$ 1) $-$ 1) $<$ length (cdddr (*body*))))
$\rightarrow$ (get (car (*pos*), *body*) = cadddr (*body*))

THEOREM: lr-proper-exprp-cadr-temps
(lr-proper-exprp (**t**, *expr*, *pnames*, *formals*, *temps*, *table*)
$\wedge$ ((car (*expr*) = S-TEMP-EVAL) $\vee$ (car (*expr*) = S-TEMP-TEST)))
$\rightarrow$ lr-proper-exprp (**t**, cadr (*expr*), *pnames*, *formals*, *temps*, *table*)

THEOREM: lessp-plus-lr-p-c-size-lr-p-pc-1-temps
(good-posp1 (*pos*, cadr (*body*))
 ∧ listp (*body*)
 ∧ ((car (*body*) = S-TEMP-EVAL) ∨ (car (*body*) = S-TEMP-TEST))
 ∧ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*))
→ (lr-p-c-size (**t**, cadr (*body*))
  ≮ (lr-p-pc-1 (cadr (*body*), *pos*)
    + lr-p-c-size (**t**, cur-expr (*pos*, cadr (*body*)))))

THEOREM: firstn-lr-p-c-size-restn-lr-p-pc-1-comp-body-1-helper-6
(listp (*body*)
 ∧ (car (*body*) ≠ 'if)
 ∧ (car (*body*) ≠ S-TEMP-FETCH)
 ∧ (car (*body*) ≠ S-TEMP-EVAL)
 ∧ (car (*body*) ≠ S-TEMP-TEST)
 ∧ (car (*body*) ≠ 'quote)
 ∧ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*)
 ∧ (*n* ≯ 0))
→ (lr-p-c-size-list (length (*body*) − 1, *body*)
  ≮ (lr-p-c-size-list (*n* − 1, *body*) + lr-p-pc-1 (get (*n*, *body*), *pos*)))

DEFINITION:
induct-hint-10 (*n*, *l*, *x*)
= **if** ¬ listp (*l*) **then t**
  elseif *n* ≃ 0 **then t**
  elseif listp (cdr (*l*))
  then induct-hint-10 (*n* − 1, cdr (*l*), *x* + lr-p-c-size (**t**, cadr (*l*)))
  else t endif

THEOREM: lr-p-c-size-list-car-opener
((*n* ≯ 0) ∧ (*n* < length (*body*)))
→ (lr-p-c-size-list (*n*, *body*)
  = (lr-p-c-size (**t**, cadr (*body*))
    + lr-p-c-size-list (*n* − 1, cdr (*body*))))

THEOREM: restn-comp-body-1-list-fact
((lr-p-c-size (**t**, get (*m*, cdr (*body*))) ≮ *j*)
 ∧ (*m* < length (cdr (*body*)))
 ∧ (*m* ∈ **N**)
 ∧ (*n* ∈ **N**)
 ∧ (*j* ∈ **N**))
→ (restn (lr-p-c-size-list (*m*, *body*) + *j*,
      comp-body-1 ('list, cdr (*body*), *n*))
  = restn (*j*,
      comp-body-1 ('list,

$\text{restn}\,(m,\ \text{cdr}\,(\mathit{body})),$
$n + \text{lr-p-c-size-list}\,(m,\ \mathit{body}))))$

EVENT: Disable lr-p-c-size-list-car-opener.

THEOREM: firstn-restn-small-enough-cdr-comp-body-1-list

$$(\text{listp}\,(\mathit{body}) \land (\text{lr-p-c-size}\,(\mathbf{t},\ \text{car}\,(\mathit{body})) \not< (j + k)))$$
$$\rightarrow\ (\text{firstn}\,(j,\ \text{restn}\,(k,\ \text{comp-body-1}\,(\texttt{'list},\ \mathit{body},\ n)))$$
$$=\ \text{firstn}\,(j,\ \text{restn}\,(k,\ \text{comp-body-1}\,(\mathbf{t},\ \text{car}\,(\mathit{body}),\ n))))$$

THEOREM: firstn-lr-p-c-size-restn-lr-p-pc-1-comp-body-1-helper-7

$$\begin{aligned}
&(\text{listp}\,(\mathit{body}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \texttt{'if}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \text{S-TEMP-FETCH}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \text{S-TEMP-EVAL}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \text{S-TEMP-TEST}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \texttt{'quote}) \\
&\land\ (n \in \mathbf{N}) \\
&\land\ \text{lr-proper-exprp}\,(\mathbf{t},\ \mathit{body},\ \mathit{pnames},\ \mathit{formals},\ \mathit{temps},\ \mathit{table}) \\
&\land\ (m \not\simeq \texttt{0}) \\
&\land\ (m < \text{length}\,(\mathit{body})) \\
&\land\ \text{good-posp1}\,(\mathit{pos},\ \text{get}\,(m,\ \mathit{body}))) \\
\rightarrow\ &(\text{firstn}\,(\text{lr-p-c-size}\,(\mathbf{t},\ \text{cur-expr}\,(\mathit{pos},\ \text{get}\,(m,\ \mathit{body}))), \\
&\qquad \text{restn}\,(\text{lr-p-c-size-list}\,(m - 1,\ \mathit{body}) \\
&\qquad\qquad + \ \text{lr-p-pc-1}\,(\text{get}\,(m,\ \mathit{body}),\ \mathit{pos}), \\
&\qquad\qquad \text{comp-body-1}\,(\texttt{'list},\ \text{cdr}\,(\mathit{body}),\ n))) \\
&= \ \text{firstn}\,(\text{lr-p-c-size}\,(\mathbf{t},\ \text{cur-expr}\,(\mathit{pos},\ \text{get}\,(m,\ \mathit{body}))), \\
&\qquad \text{restn}\,(\text{lr-p-pc-1}\,(\text{get}\,(m,\ \mathit{body}),\ \mathit{pos}), \\
&\qquad\qquad \text{comp-body-1}\,(\mathbf{t}, \\
&\qquad\qquad\qquad \text{get}\,(m,\ \mathit{body}), \\
&\qquad\qquad\qquad n + \text{lr-p-c-size-list}\,(m - 1,\ \mathit{body})))))
\end{aligned}$$

THEOREM: firstn-lr-p-c-size-restn-lr-p-pc-1-comp-body-1-helper-8

$$\begin{aligned}
&(\text{listp}\,(\mathit{body}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \texttt{'if}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \text{S-TEMP-FETCH}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \text{S-TEMP-EVAL}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \text{S-TEMP-TEST}) \\
&\land\ (\text{car}\,(\mathit{body}) \neq \texttt{'quote}) \\
&\land\ \text{lr-proper-exprp}\,(\mathbf{t},\ \mathit{body},\ \mathit{pnames},\ \mathit{formals},\ \mathit{temps},\ \mathit{table}) \\
&\land\ (n \not\simeq \texttt{0}) \\
&\land\ (n < \text{length}\,(\mathit{body})) \\
&\land\ \text{good-posp1}\,(\mathit{pos},\ \text{get}\,(n,\ \mathit{body}))) \\
\rightarrow\ &((\text{lr-p-c-size-list}\,(\text{length}\,(\mathit{body}) - 1,\ \mathit{body})
\end{aligned}$$

$-$ (lr-p-c-size-list ($n - 1$, *body*) + lr-p-pc-1 (get (n, *body*), *pos*)))
$\not<$ lr-p-c-size (**t**, cur-expr (*pos*, get (n, *body*))))

THEOREM: firstn-lr-p-c-size-restn-lr-p-pc-1-comp-body-1
(good-posp1 (*pos*, *body*)
$\wedge$ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*)
$\wedge$ ($n \in \mathbf{N}$))
$\rightarrow$ (firstn (lr-p-c-size (**t**, cur-expr (*pos*, *body*)),
restn (lr-p-pc-1 (*body*, *pos*), comp-body-1 (**t**, *body*, n)))
$=$ comp-body-1 (**t**, cur-expr (*pos*, *body*), n + lr-p-pc-1 (*body*, *pos*)))

EVENT: Disable firstn-lr-p-c-size-restn-lr-p-pc-1-comp-body-1-helper-5.

THEOREM: not-lessp-lr-p-c-size-flag-t-1
lr-p-c-size (**t**, *body1*) $\not<$ `1`

THEOREM: not-lessp-x-x
($x < x$) = **f**

THEOREM: get-plus
get ($x + y$, *list*) = get (y, restn (x, *list*))

EVENT: Disable get-plus.

THEOREM: get-firstn-different-lists
(($k < n$) $\wedge$ (firstn (n, *list1*) = firstn (n, *list2*)))
$\rightarrow$ (get (k, *list1*) = get (k, *list2*))

THEOREM: unlabel-list-label
unlabel (list (`'dl`, *lab*, *comment*, *instr*)) = *instr*

THEOREM: legal-labelp-label-make-label
legal-labelp (list (`'dl`, lr-make-label (n), *comment*, *instr*))

THEOREM: lr-make-label-not-numberp
($n \notin \mathbf{N}$) $\rightarrow$ (lr-make-label (n) = lr-make-label (`0`))

DEFINITION:
induct-hint-9 (m, *instrs*, n)
$=$ **if** listp (*instrs*) **then** induct-hint-9 ($m - 1$, cdr (*instrs*), $1 + n$)
else t endif

THEOREM: get-label-instrs
($m <$ length (*instrs*))
$\rightarrow$ (get (m, label-instrs (*instrs*, n))
$=$ list (`'dl`, lr-make-label ($n + m$), **nil**, get (m, *instrs*)))

EVENT: Disable lr-make-label-not-numberp.

THEOREM: get-append
get (*n*, append (*x*, *y*))
= **if** *n* < length (*x*) **then** get (*n*, *x*)
else get (*n* − length (*x*), *y*) **endif**

EVENT: Disable get-append.

THEOREM: get-lr-p-c-size-lessp-lr-p-c-size-comp-body-1
(good-posp1 (*pos*, *body*)
∧ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*)
∧ (*n* ∈ **N**)
∧ (*m* < lr-p-c-size (**t**, cur-expr (*pos*, *body*))))
→ (get (lr-p-pc-1 (*body*, *pos*) + *m*, comp-body-1 (**t**, *body*, *n*))
= get (*m*,
comp-body-1 (**t**,
cur-expr (*pos*, *body*),
n + lr-p-pc-1 (*body*, *pos*))))

THEOREM: get-lr-p-pc-1-comp-body-1-cur-expr-comp-body
(good-posp1 (offset (p-pc (*l*)), program-body (*prog*))
∧ listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) = `'quote`)
∧ lr-programs-properp (*l*, *table*)
∧ (*prog* = p-current-program (*l*)))
→ (get (lr-p-pc-1 (program-body (*prog*), offset (p-pc (*l*))),
comp-body (program-body (*prog*)))
= list (`'dl`,
lr-make-label (lr-p-pc-1 (program-body (*prog*),
offset (p-pc (*l*)))),
nil,
list (`'push-constant`, cadr (lr-expr (*l*))))))

THEOREM: get-lr-p-pc-1-comp-body-1-quote
(good-posp1 (offset (p-pc (*l*)), program-body (car (p-prog-segment (*l*))))
∧ listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) = `'quote`)
∧ lr-programs-properp (*l*, *table*)
∧ (area-name (p-pc (*l*)) = caar (p-prog-segment (*l*))))
→ (get (lr-p-pc-1 (program-body (car (p-prog-segment (*l*))), offset (p-pc (*l*))),
comp-body-1 (**t**, program-body (car (p-prog-segment (*l*))), `0`))
= list (`'push-constant`, cadr (lr-expr (*l*)))))

THEOREM: proper-p-temp-stkp-p-temp-stk-lr-push-tstk-quote
(lr-programs-properp (l, *table*)
 ∧ proper-p-prog-segmentp (comp-programs (p-prog-segment (l)), lr->p (l))
 ∧ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧ listp (lr-expr (l))
 ∧ (car (lr-expr (l)) = 'quote)
 ∧ proper-p-temp-stkp (p-temp-stk (l), lr->p (l)))
→ proper-p-temp-stkp (p-temp-stk (lr-push-tstk (l, cadr (lr-expr (l)))),
lr->p (l))

THEOREM: proper-p-statep-lr-push-tstk-quote
(proper-p-statep (lr->p (l))
 ∧ listp (lr-expr (l))
 ∧ (car (lr-expr (l)) = 'quote)
 ∧ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧ lr-programs-properp (l, *table*)
 ∧ (p-psw (lr-push-tstk (l, cadr (lr-expr (l)))) = 'run))
→ proper-p-statep (lr->p (lr-push-tstk (l, cadr (lr-expr (l))))))

THEOREM: good-posp-dv-1-funcall-lr-expr
(listp (lr-expr (l))
 ∧ (car (lr-expr (l)) ≠ 'if)
 ∧ (car (lr-expr (l)) ≠ S-TEMP-EVAL)
 ∧ (car (lr-expr (l)) ≠ S-TEMP-TEST)
 ∧ (car (lr-expr (l)) ≠ S-TEMP-FETCH)
 ∧ (car (lr-expr (l)) ≠ 'quote)
 ∧ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l))))
→ good-posp ('list,
dv (offset (p-pc (l)), 1),
program-body (p-current-program (l)))

THEOREM: plistp-pairlist
plistp (pairlist (x, y))

THEOREM: all-p-objectps-append
plistp (*lst1*)
→ (all-p-objectps (append (*lst1*, *lst2*), p)
= (all-p-objectps (*lst1*, p) ∧ all-p-objectps (*lst2*, p)))

THEOREM: all-p-objectps-reverse
plistp (*lst*) → (all-p-objectps (reverse (*lst*), p) = all-p-objectps (*lst*, p))

THEOREM: plistp-first-n
plistp (first-n (n, *list*))

THEOREM: proper-p-temp-stkp-all-p-objectps
proper-p-temp-stkp (*temp-stk*, *p*) → all-p-objectps (*temp-stk*, *p*)

THEOREM: all-p-objectps-first-n
((length (*lst*) ≮ *n*) ∧ all-p-objectps (*lst*, *p*))
→ all-p-objectps (first-n (*n*, *lst*), *p*)

THEOREM: strip-cars-append
strip-cars (append (*x*, *y*)) = append (strip-cars (*x*), strip-cars (*y*))

EVENT: Disable strip-cars-append.

THEOREM: strip-cars-pairlist
strip-cars (pairlist (*x*, *y*)) = plist (*x*)

EVENT: Disable strip-cars-pairlist.

THEOREM: strip-cars-pair-temps-with-initial-values
strip-cars (pair-temps-with-initial-values (*temp-var-decls*))
= strip-cars (*temp-var-decls*)

THEOREM: length-popn
(length (*list*) ≮ *n*) → (length (popn (*n*, *list*)) = (length (*list*) − *n*))

THEOREM: proper-p-temp-stkp-popn
((length (*temp-stk*) ≮ *n*) ∧ proper-p-temp-stkp (*temp-stk*, *p*))
→ proper-p-temp-stkp (popn (*n*, *temp-stk*), *p*)

THEOREM: proper-p-prog-segmentp-length-program-body
(proper-p-prog-segmentp (*prog-segment*, *p*) ∧ definedp (*name*, *prog-segment*))
→ listp (program-body (assoc (*name*, *prog-segment*)))

THEOREM: ret-pc-make-p-call-frame
ret-pc (make-p-call-frame (*f-vars*, *temp-stk*, *temp-var-dcls*, *ret-pc*)) = *ret-pc*

THEOREM: bindings-make-p-call-frame
bindings (make-p-call-frame (*f-vars*, *temp-stk*, *temp-var-dcls*, *ret-pc*))
= append (pair-formal-vars-with-actuals (*f-vars*, *temp-stk*),
pair-temps-with-initial-values (*temp-var-dcls*))

THEOREM: cddr-nil-make-p-call-frame
cddr (make-p-call-frame (*f-vars*, *temp-stk*, *temp-var-dcls*, *ret-pc*)) = **nil**

THEOREM: listp-cdr-make-p-call-frame
listp (cdr (make-p-call-frame (*f-vars*, *temp-stk*, *temp-var-dcls*, *ret-pc*)))

THEOREM: length-pairlist
length (pairlist (x, y)) = length (x)

THEOREM: length-pair-temps-with-initial-values
length (pair-temps-with-initial-values (*temp-var-dcls*))
= length (*temp-var-dcls*)

THEOREM: not-proper-p-statep-not-listp-p-ctrl-stk
($\neg$ listp (p-ctrl-stk (l))) $\rightarrow$ ($\neg$ proper-p-statep (lr->p (l)))

THEOREM: proper-p-statep-bad-type-1
((fetch (car (p-temp-stk (l)), p-data-segment (l))
$\neq$ list (type (fetch (car (p-temp-stk (l)), p-data-segment (l))),
untag (fetch (car (p-temp-stk (l)), p-data-segment (l)))))
$\wedge$ adpp (untag (car (p-temp-stk (l))), p-data-segment (l)))
$\rightarrow$ ($\neg$ proper-p-statep (lr->p (l)))

THEOREM: p-good-resultp-run-car
(proper-p-statep (lr->p (l))
$\wedge$ (p-psw (l) = `'run`)
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (l))))))
= `'(call car)`))
$\rightarrow$ p-good-resultp (p (p-set-pc (lr->p (l), *pc*),
p-car-clock (p-set-pc (lr->p (l), *pc*))),
p-data-segment (l),
if fetch (top (p-temp-stk (l)), p-data-segment (l))
= tag (`'nat`, LR-CONS-TAG)
then cons (fetch (add-addr (top (p-temp-stk (l)),
LR-CAR-OFFSET),
p-data-segment (l)),
cdr (p-temp-stk (l)))
else cons (LR-0-ADDR, cdr (p-temp-stk (l))) **endif**,
p-ctrl-stk (l),
add-addr (*pc*, `1`))

THEOREM: p-good-resultp-run-cdr
(proper-p-statep (lr->p (l))
$\wedge$ (p-psw (l) = `'run`)
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (l))))))

```
      =  '(call cdr)))
→   p-good-resultp (p (p-set-pc (lr->p (l), pc),
                         p-cdr-clock (p-set-pc (lr->p (l), pc))),
                      p-data-segment (l),
                      if fetch (top (p-temp-stk (l)), p-data-segment (l))
                        =  tag ('nat, LR-CONS-TAG)
                      then cons (fetch (add-addr (top (p-temp-stk (l)),
                                                      LR-CDR-OFFSET),
                                                 p-data-segment (l)),
                                 cdr (p-temp-stk (l)))
                      else cons (LR-0-ADDR, cdr (p-temp-stk (l))) endif,
                      p-ctrl-stk (l),
                      add-addr (pc, 1))

THEOREM: p-good-resultp-run-listp
 (proper-p-statep (lr->p (l))
  ∧  (p-psw (l) = 'run)
  ∧  lr-programs-properp (l, table)
  ∧  (unlabel (get (offset (pc),
                   program-body (assoc (area-name (pc),
                                        p-prog-segment (lr->p (l))))))
      =  '(call listp)))
→   p-good-resultp (p (p-set-pc (lr->p (l), pc),
                         p-listp-clock (p-set-pc (lr->p (l), pc))),
                      p-data-segment (l),
                      if fetch (car (p-temp-stk (l)), p-data-segment (l))
                        =  tag ('nat, LR-CONS-TAG)
                      then cons (LR-T-ADDR, cdr (p-temp-stk (l)))
                      else cons (LR-F-ADDR, cdr (p-temp-stk (l))) endif,
                      p-ctrl-stk (l),
                      add-addr (pc, 1))

THEOREM: p-good-resultp-run-nlistp
 (proper-p-statep (lr->p (l))
  ∧  (p-psw (l) = 'run)
  ∧  lr-programs-properp (l, table)
  ∧  (unlabel (get (offset (pc),
                   program-body (assoc (area-name (pc),
                                        p-prog-segment (lr->p (l))))))
      =  '(call nlistp)))
→   p-good-resultp (p (p-set-pc (lr->p (l), pc),
                         p-nlistp-clock (p-set-pc (lr->p (l), pc))),
                      p-data-segment (l),
                      if fetch (car (p-temp-stk (l)), p-data-segment (l))
```

```
                    =   tag ('nat, LR-CONS-TAG)
                  then cons (LR-F-ADDR, cdr (p-temp-stk (l)))
                  else cons (LR-T-ADDR, cdr (p-temp-stk (l))) endif,
                  p-ctrl-stk (l),
                  add-addr (pc, 1))
```

THEOREM: p-good-resultp-run-truep

```
 (proper-p-statep (lr->p (l))
  ∧   (p-psw (l) = 'run)
  ∧   lr-programs-properp (l, table)
  ∧   (unlabel (get (offset (pc),
                    program-body (assoc (area-name (pc),
                                         p-prog-segment (lr->p (l))))))
       =   '(call truep)))
  →   p-good-resultp (p (p-set-pc (lr->p (l), pc),
                         p-truep-clock (p-set-pc (lr->p (l), pc))),
                      p-data-segment (l),
                      if fetch (car (p-temp-stk (l)), p-data-segment (l))
                        =   tag ('nat, LR-TRUE-TAG)
                      then cons (LR-T-ADDR, cdr (p-temp-stk (l)))
                      else cons (LR-F-ADDR, cdr (p-temp-stk (l))) endif,
                      p-ctrl-stk (l),
                      add-addr (pc, 1))
```

EVENT: Disable proper-p-statep-bad-type-1.

THEOREM: p-good-resultp-run-cons

```
 (proper-p-statep (lr->p (l))
  ∧   (p-psw (l) = 'run)
  ∧   lr-programs-properp (l, table)
  ∧   (unlabel (get (offset (pc),
                    program-body (assoc (area-name (pc),
                                         p-prog-segment (lr->p (l))))))
       =   '(call cons)))
  →   p-good-resultp (p (p-set-pc (lr->p (l), pc),
                         p-cons-clock (p-set-pc (lr->p (l), pc))),
                      deposit (fetch (add-addr (fetch (LR-FP-ADDR,
                                                       p-data-segment (l)),
                                                  LR-REF-COUNT-OFFSET),
                                               p-data-segment (l)),
                                      LR-FP-ADDR,
                                      deposit-a-list (list (tag ('nat, LR-CONS-TAG),
                                                            tag ('nat, 1),
                                                            top1 (p-temp-stk (l)),
```

top (p-temp-stk (*l*))),
fetch (LR-FP-ADDR,
p-data-segment (*l*)),
p-data-segment (*l*))),
cons (fetch (LR-FP-ADDR, p-data-segment (*l*)),
cddr (p-temp-stk (*l*))),
p-ctrl-stk (*l*),
add-addr (*pc*, `1`))

THEOREM: p-objectp-bad-type
(*object* ≠ list (type (*object*), untag (*object*))) → (¬ p-objectp (*object*, *p*))

THEOREM: proper-p-statep-bad-type-2
((car (p-temp-stk (*l*))
≠ list (type (car (p-temp-stk (*l*))), untag (car (p-temp-stk (*l*)))))
∧ listp (p-temp-stk (*l*)))
→ (¬ proper-p-statep (lr->p (*l*)))

THEOREM: p-good-resultp-run-falsep
(proper-p-statep (lr->p (*l*))
∧ (p-psw (*l*) = `'run`)
∧ lr-programs-properp (*l*, *table*)
∧ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (*l*))))))
= `'(call falsep)`))
→ p-good-resultp (p (p-set-pc (lr->p (*l*), *pc*),
p-falsep-clock (p-set-pc (lr->p (*l*), *pc*))),
p-data-segment (*l*),
if car (p-temp-stk (*l*)) = LR-F-ADDR
then cons (LR-T-ADDR, cdr (p-temp-stk (*l*)))
else cons (LR-F-ADDR, cdr (p-temp-stk (*l*))) **endif**,
p-ctrl-stk (*l*),
add-addr (*pc*, `1`))

EVENT: Disable proper-p-statep-bad-type-2.

THEOREM: p-good-resultp-run-false
(proper-p-statep (lr->p (*l*))
∧ (p-psw (*l*) = `'run`)
∧ lr-programs-properp (*l*, *table*)
∧ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (*l*))))))

```
      =  '(call false)))
→   p-good-resultp (p (p-set-pc (lr->p (l), pc),
                          p-false-clock (p-set-pc (lr->p (l), pc))),
                       p-data-segment (l),
                       cons (LR-F-ADDR, p-temp-stk (l)),
                       p-ctrl-stk (l),
                       add-addr (pc, 1))

THEOREM: p-good-resultp-run-true
 (proper-p-statep (lr->p (l))
 ∧   (p-psw (l) = 'run)
 ∧   lr-programs-properp (l, table)
 ∧   (unlabel (get (offset (pc),
                   program-body (assoc (area-name (pc),
                                        p-prog-segment (lr->p (l))))))
      =  '(call true)))
→   p-good-resultp (p (p-set-pc (lr->p (l), pc),
                          p-true-clock (p-set-pc (lr->p (l), pc))),
                       p-data-segment (l),
                       cons (LR-T-ADDR, p-temp-stk (l)),
                       p-ctrl-stk (l),
                       add-addr (pc, 1))

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-car
 (proper-p-statep (lr->p (l))
 ∧   (p-psw (l) = 'run)
 ∧   (p-psw (p (p-set-pc (lr->p (l), pc), p-car-clock (p-set-pc (lr->p (l), pc))))
      =  'run)
 ∧   lr-programs-properp (l, table)
 ∧   (unlabel (get (offset (pc),
                   program-body (assoc (area-name (pc),
                                        p-prog-segment (lr->p (l))))))
      =  '(call car)))
→   ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
                         p-car-clock (p-set-pc (lr->p (l), pc))))
      =  if fetch (top (p-temp-stk (l)), p-data-segment (l))
           =  tag ('nat, LR-CONS-TAG)
         then cons (fetch (add-addr (top (p-temp-stk (l)), LR-CAR-OFFSET),
                           p-data-segment (l)),
                    cdr (p-temp-stk (l)))
         else cons (LR-0-ADDR, cdr (p-temp-stk (l))) endif)
    ∧   (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
                         p-car-clock (p-set-pc (lr->p (l), pc))))
         =  p-ctrl-stk (l))
```

```
    ∧   (p-data-segment (p (p-set-pc (lr->p (l), pc),
                                     p-car-clock (p-set-pc (lr->p (l), pc))))
          =   p-data-segment (l)))

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-cdr
 (proper-p-statep (lr->p (l))
  ∧   (p-psw (l) = 'run)
  ∧   (p-psw (p (p-set-pc (lr->p (l), pc), p-cdr-clock (p-set-pc (lr->p (l), pc))))
        =   'run)
  ∧   lr-programs-properp (l, table)
  ∧   (unlabel (get (offset (pc),
                     program-body (assoc (area-name (pc),
                                          p-prog-segment (lr->p (l))))))
        =   '(call cdr)))
 →   ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
                       p-cdr-clock (p-set-pc (lr->p (l), pc))))
       =   if fetch (top (p-temp-stk (l)), p-data-segment (l))
             =   tag ('nat, LR-CONS-TAG)
           then cons (fetch (add-addr (top (p-temp-stk (l)), LR-CDR-OFFSET),
                             p-data-segment (l)),
                      cdr (p-temp-stk (l)))
           else cons (LR-0-ADDR, cdr (p-temp-stk (l))) endif)
      ∧   (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
                          p-cdr-clock (p-set-pc (lr->p (l), pc))))
            =   p-ctrl-stk (l))
      ∧   (p-data-segment (p (p-set-pc (lr->p (l), pc),
                              p-cdr-clock (p-set-pc (lr->p (l), pc))))
            =   p-data-segment (l)))

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-cons
 let fp-addr be fetch (LR-FP-ADDR, p-data-segment (l))
 in
 (proper-p-statep (lr->p (l))
  ∧   (p-psw (l) = 'run)
  ∧   (p-psw (p (p-set-pc (lr->p (l), pc),
                 p-cons-clock (p-set-pc (lr->p (l), pc))))
        =   'run)
  ∧   lr-programs-properp (l, table)
  ∧   (unlabel (get (offset (pc),
                     program-body (assoc (area-name (pc),
                                          p-prog-segment (lr->p (l))))))
        =   '(call cons)))
 →   ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
                       p-cons-clock (p-set-pc (lr->p (l), pc))))
```

```
    =   cons (fetch (LR-FP-ADDR, p-data-segment (l)),
             cddr (p-temp-stk (l))))
∧   (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
                     p-cons-clock (p-set-pc (lr->p (l), pc))))
    =   p-ctrl-stk (l))
∧   (p-data-segment (p (p-set-pc (lr->p (l), pc),
                              p-cons-clock (p-set-pc (lr->p (l), pc))))
    =   deposit (fetch (add-addr (fetch (LR-FP-ADDR,
                                         p-data-segment (l)),
                                  LR-REF-COUNT-OFFSET),
                        p-data-segment (l)),
                 LR-FP-ADDR,
                 deposit-a-list (list (tag ('nat,
                                            LR-CONS-TAG),
                                       tag ('nat, 1),
                                       top1 (p-temp-stk (l)),
                                       top (p-temp-stk (l))),
                                 fetch (LR-FP-ADDR,
                                        p-data-segment (l)),
                                 p-data-segment (l))))) endlet
```

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-false

```
(proper-p-statep (lr->p (l))
 ∧   (p-psw (l) = 'run)
 ∧   (p-psw (p (p-set-pc (lr->p (l), pc), p-false-clock (p-set-pc (lr->p (l), pc))))
     =   'run)
 ∧   lr-programs-properp (l, table)
 ∧   (unlabel (get (offset (pc),
                    program-body (assoc (area-name (pc),
                                         p-prog-segment (lr->p (l))))))
     =   '(call false)))
→   ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
                       p-false-clock (p-set-pc (lr->p (l), pc))))
     =   cons (LR-F-ADDR, p-temp-stk (l)))
    ∧   (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
                              p-false-clock (p-set-pc (lr->p (l), pc))))
        =   p-ctrl-stk (l))
    ∧   (p-data-segment (p (p-set-pc (lr->p (l), pc),
                                  p-false-clock (p-set-pc (lr->p (l), pc))))
        =   p-data-segment (l)))
```

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-falsep

```
(proper-p-statep (lr->p (l))
 ∧   (p-psw (l) = 'run)
```

```
 ∧   (p-psw (p (p-set-pc (lr->p (l), pc), p-falsep-clock (p-set-pc (lr->p (l), pc))))
       =   'run)
 ∧   lr-programs-properp (l, table)
 ∧   (unlabel (get (offset (pc),
                    program-body (assoc (area-name (pc),
                                         p-prog-segment (lr->p (l))))))
       =   '(call falsep)))
 →   ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
                      p-falsep-clock (p-set-pc (lr->p (l), pc))))
       =   if car (p-temp-stk (l)) = LR-F-ADDR
           then cons (LR-T-ADDR, cdr (p-temp-stk (l)))
           else cons (LR-F-ADDR, cdr (p-temp-stk (l))) endif)
       ∧   (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
                           p-falsep-clock (p-set-pc (lr->p (l), pc))))
           =   p-ctrl-stk (l))
       ∧   (p-data-segment (p (p-set-pc (lr->p (l), pc),
                               p-falsep-clock (p-set-pc (lr->p (l), pc))))
           =   p-data-segment (l)))

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-listp
 (proper-p-statep (lr->p (l))
 ∧   (p-psw (l) = 'run)
 ∧   (p-psw (p (p-set-pc (lr->p (l), pc), p-listp-clock (p-set-pc (lr->p (l), pc))))
       =   'run)
 ∧   lr-programs-properp (l, table)
 ∧   (unlabel (get (offset (pc),
                    program-body (assoc (area-name (pc),
                                         p-prog-segment (lr->p (l))))))
       =   '(call listp)))
 →   ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
                      p-listp-clock (p-set-pc (lr->p (l), pc))))
       =   if fetch (car (p-temp-stk (l)), p-data-segment (l))
             =   tag ('nat, LR-CONS-TAG)
           then cons (LR-T-ADDR, cdr (p-temp-stk (l)))
           else cons (LR-F-ADDR, cdr (p-temp-stk (l))) endif)
       ∧   (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
                           p-listp-clock (p-set-pc (lr->p (l), pc))))
           =   p-ctrl-stk (l))
       ∧   (p-data-segment (p (p-set-pc (lr->p (l), pc),
                               p-listp-clock (p-set-pc (lr->p (l), pc))))
           =   p-data-segment (l)))

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-nlistp
 (proper-p-statep (lr->p (l))
```

$\wedge$ (p-psw (l) = 'run)
$\wedge$ (p-psw (p (p-set-pc (lr->p (l), pc), p-nlistp-clock (p-set-pc (lr->p (l), pc))))
= 'run)
$\wedge$ lr-programs-properp (l, $table$)
$\wedge$ (unlabel (get (offset (pc),
program-body (assoc (area-name (pc),
p-prog-segment (lr->p (l))))))
= '(call nlistp)))
$\rightarrow$ ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
p-nlistp-clock (p-set-pc (lr->p (l), pc))))
= **if** fetch (car (p-temp-stk (l)), p-data-segment (l))
= tag ('nat, LR-CONS-TAG)
then cons (LR-F-ADDR, cdr (p-temp-stk (l)))
else cons (LR-T-ADDR, cdr (p-temp-stk (l))) **endif**)
$\wedge$ (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
p-nlistp-clock (p-set-pc (lr->p (l), pc))))
= p-ctrl-stk (l))
$\wedge$ (p-data-segment (p (p-set-pc (lr->p (l), pc),
p-nlistp-clock (p-set-pc (lr->p (l), pc))))
= p-data-segment (l)))

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-true
(proper-p-statep (lr->p (l))
$\wedge$ (p-psw (l) = 'run)
$\wedge$ (p-psw (p (p-set-pc (lr->p (l), pc), p-true-clock (p-set-pc (lr->p (l), pc))))
= 'run)
$\wedge$ lr-programs-properp (l, $table$)
$\wedge$ (unlabel (get (offset (pc),
program-body (assoc (area-name (pc),
p-prog-segment (lr->p (l))))))
= '(call true)))
$\rightarrow$ ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
p-true-clock (p-set-pc (lr->p (l), pc))))
= cons (LR-T-ADDR, p-temp-stk (l)))
$\wedge$ (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
p-true-clock (p-set-pc (lr->p (l), pc))))
= p-ctrl-stk (l))
$\wedge$ (p-data-segment (p (p-set-pc (lr->p (l), pc),
p-true-clock (p-set-pc (lr->p (l), pc))))
= p-data-segment (l)))

THEOREM: p-temp-stk-p-ctrl-stk-p-data-segment-run-truep
(proper-p-statep (lr->p (l))
$\wedge$ (p-psw (l) = 'run)

$\wedge$ (p-psw (p (p-set-pc (lr->p (l), pc), p-truep-clock (p-set-pc (lr->p (l), pc))))
= 'run)
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (unlabel (get (offset (pc),
program-body (assoc (area-name (pc),
p-prog-segment (lr->p (l))))))
= '(call truep)))
$\rightarrow$ ((p-temp-stk (p (p-set-pc (lr->p (l), pc),
p-truep-clock (p-set-pc (lr->p (l), pc))))
= **if** fetch (car (p-temp-stk (l)), p-data-segment (l))
= tag ('nat, LR-TRUE-TAG)
then cons (LR-T-ADDR, cdr (p-temp-stk (l)))
else cons (LR-F-ADDR, cdr (p-temp-stk (l))) **endif**)
$\wedge$ (p-ctrl-stk (p (p-set-pc (lr->p (l), pc),
p-truep-clock (p-set-pc (lr->p (l), pc))))
= p-ctrl-stk (l))
$\wedge$ (p-data-segment (p (p-set-pc (lr->p (l), pc),
p-truep-clock (p-set-pc (lr->p (l), pc))))
= p-data-segment (l)))

THEOREM: get-last-funcall-cur-expr
(listp (*expr*)
$\wedge$ (car (*expr*) $\neq$ 'if)
$\wedge$ (car (*expr*) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (*expr*) $\neq$ S-TEMP-TEST)
$\wedge$ (car (*expr*) $\neq$ S-TEMP-FETCH)
$\wedge$ (car (*expr*) $\neq$ 'quote))
$\rightarrow$ (get (lr-p-c-size-list (length (*expr*) $-$ 1, *expr*), comp-body-1 (**t**, *expr*, n))
= **if** definedp (car (*expr*), P-RUNTIME-SUPPORT-PROGRAMS)
then list ('call, car (*expr*))
else list ('call, user-fname (car (*expr*))) **endif**)

THEOREM: not-listp-p-prog-segment-lr-expr
($\neg$ listp (p-prog-segment (l))) $\rightarrow$ ($\neg$ listp (lr-expr (l)))

THEOREM: get-offset-return-pc-program-body-assoc-comp-programs
(good-posp1 (offset (p-pc (l)),
program-body (assoc (area-name (p-pc (l)), p-prog-segment (l)))))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-TEST)
$\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-FETCH)
$\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
$\wedge$ listp (lr-expr (l)))

```
→  (get (offset (lr-return-pc (l)),
         program-body (assoc (area-name (p-pc (l)),
                                        comp-programs (p-prog-segment (l))))))
   =  list ('dl,
            lr-make-label (offset (lr-return-pc (l))),
            nil,
            if definedp (car (lr-expr (l)), P-RUNTIME-SUPPORT-PROGRAMS)
            then list ('call, car (lr-expr (l)))
            else list ('call, user-fname (car (lr-expr (l)))) endif))
```

THEOREM: listp-p-temp-stk-proper-ctrl-stk-p-run-subr

```
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  lr-programs-properp (new-l, table)
 ∧  lr-programs-properp (l, table)
 ∧  listp (lr-expr (l))
 ∧  proper-p-statep (lr->p (new-l))
 ∧  (p-psw (new-l) = 'run)
 ∧  (p-psw (p-run-subr (car (lr-expr (l)),
                        p-set-pc (lr->p (new-l), lr-return-pc (l))))
     =  'run)
 ∧  (p-prog-segment (l) = p-prog-segment (new-l)))
→  (listp (p-temp-stk (p-run-subr (car (lr-expr (l)),
                                   p-set-pc (lr->p (new-l), lr-return-pc (l))))))
    ∧  (p-ctrl-stk (p-run-subr (car (lr-expr (l)),
                                p-set-pc (lr->p (new-l), lr-return-pc (l))))
        =  p-ctrl-stk (new-l)))
```

THEOREM: listp-p-temp-stk-proper-ctrl-stk-lr-apply-subr

```
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  lr-programs-properp (l, table)
 ∧  lr-programs-properp (new-l, table)
 ∧  listp (lr-expr (l))
 ∧  subrp (car (lr-expr (l)))
 ∧  (p-psw (lr-apply-subr (l, new-l)) = 'run)
 ∧  proper-p-statep (lr->p (new-l))
 ∧  (p-psw (new-l) = 'run)
 ∧  (p-prog-segment (l) = p-prog-segment (new-l)))
→  (listp (p-temp-stk (lr-apply-subr (l, new-l)))
    ∧  (p-ctrl-stk (lr-apply-subr (l, new-l)) = p-ctrl-stk (new-l)))
```

THEOREM: cur-expr-nlistp-pos

```
(pos ≃ nil) → (cur-expr (pos, body) = body)
```

THEOREM: proper-p-statep-p-run-subr

```
(proper-p-statep (p) ∧ (p-psw (p-run-subr (subr, p)) = 'run))
→  proper-p-statep (p-run-subr (subr, p))
```

THEOREM: same-signature-commutative
same-signature (*x*, *y*) = same-signature (*y*, *x*)

THEOREM: same-signature-p-run-subr
(proper-p-statep (*p*)
 ∧ (p-psw (p-run-subr (*subr*, *p*)) = 'run)
 ∧ (*data-seg* = p-data-segment (*p*)))
→ same-signature (*data-seg*, p-data-segment (p-run-subr (*subr*, *p*)))

THEOREM: proper-p-framep-lr->p-similar-states
(proper-p-framep (*frame*, *name*, *p0*)
 ∧ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))
 ∧ (p-prog-segment (*p0*) = p-prog-segment (*p1*))
 ∧ (p-word-size (*p0*) = p-word-size (*p1*)))
→ proper-p-framep (*frame*, *name*, *p1*)

THEOREM: car-untag-p-pc-lr-eval
car (untag (p-pc (lr-eval (*flag*, *l*, *c*)))) = car (untag (p-pc (*l*)))

THEOREM: lessp-cdr-untag-lr-return-pc-lr-p-c-size
(good-posp1 (offset (p-pc (*l*)),
                program-body (assoc (car (untag (p-pc (*l*))), p-prog-segment (*l*))))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (subrp (car (lr-expr (*l*))) ∨ litatom (car (lr-expr (*l*))))
 ∧ (car (lr-expr (*l*)) ≠ 'if)
 ∧ (car (lr-expr (*l*)) ≠ 'quote)
 ∧ (*name* = area-name (p-pc (*l*))))
→ (cdr (untag (lr-return-pc (*l*)))
    < length (program-body (assoc (*name*,
                                   comp-programs (p-prog-segment (*l*))))))

THEOREM: proper-p-statep-lr-apply-subr-state
(proper-p-statep (lr->p (*new-l*))
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ listp (lr-expr (*l*))
 ∧ subrp (car (lr-expr (*l*)))
 ∧ (car (lr-expr (*l*)) ≠ 'if)
 ∧ (p-prog-segment (*l*) = p-prog-segment (*new-l*))
 ∧ (area-name (p-pc (*l*)) = area-name (p-pc (*new-l*))))
→ proper-p-statep (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))

THEOREM: same-signature-lr-apply-subr
(proper-p-statep (lr->p (*new-l*))
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))

$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ listp (lr-expr (l))
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (p-prog-segment (l) = p-prog-segment (*new-l*))
$\wedge$ (area-name (p-pc (l)) = area-name (p-pc (*new-l*)))
$\wedge$ (p-psw (lr-apply-subr (l, *new-l*)) = 'run)
$\wedge$ (*data-seg* = p-data-segment (*new-l*)))
$\rightarrow$ same-signature (*data-seg*, p-data-segment (lr-apply-subr (l, *new-l*)))

THEOREM: p-current-program-lr-apply-subr
((area-name (p-pc (*new-l*)) = area-name (p-pc (l)))
$\wedge$ (p-prog-segment (*new-l*) = p-prog-segment (l)))
$\rightarrow$ (p-current-program (lr-apply-subr (l, *new-l*)) = p-current-program (l))

THEOREM: p-current-program-lr-eval
p-current-program (lr-eval (*flag*, l, c)) = p-current-program (l)

THEOREM: proper-p-framep-lr-apply-subr
(proper-p-statep (lr->p (*new-l*))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ listp (lr-expr (l))
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (p-prog-segment (l) = p-prog-segment (*new-l*))
$\wedge$ (area-name (p-pc (l)) = area-name (p-pc (*new-l*)))
$\wedge$ (p-psw (lr-apply-subr (l, *new-l*)) = 'run)
$\wedge$ (*name* = area-name (p-pc (*new-l*))))
$\rightarrow$ proper-p-framep (car (p-ctrl-stk (*new-l*)),
name,
lr->p (lr-apply-subr (l, *new-l*)))

THEOREM: proper-p-statep-lr->p-lessp-ctrl-stk-size
(proper-p-statep (lr->p (l)) $\wedge$ (*max* = p-max-ctrl-stk-size (l)))
$\rightarrow$ ((*max* < p-ctrl-stk-size (p-ctrl-stk (l))) = **f**)

EVENT: Disable proper-p-statep-lr->p-lessp-ctrl-stk-size.

THEOREM: proper-p-statep-lr->p-numberp-max-ctrl-stk-size
proper-p-statep (lr->p (l)) $\rightarrow$ (p-max-ctrl-stk-size (l) $\in$ **N**)

EVENT: Disable proper-p-statep-lr->p-numberp-max-ctrl-stk-size.

THEOREM: proper-p-statep-lr->p-numberp-max-temp-stk-size
proper-p-statep (lr->p (l)) → (p-max-temp-stk-size (l) ∈ **N**)

EVENT: Disable proper-p-statep-lr->p-numberp-max-temp-stk-size.

THEOREM: proper-p-statep-lr->p-numberp-word-size
proper-p-statep (lr->p (l)) → (p-word-size (l) ∈ **N**)

EVENT: Disable proper-p-statep-lr->p-numberp-word-size.

THEOREM: proper-p-statep-lr->p-lessp-max-ctrl-stk-size
proper-p-statep (lr->p (l))
→ ((p-max-ctrl-stk-size (l) < exp (2, p-word-size (l))) = **t**)

EVENT: Disable proper-p-statep-lr->p-lessp-max-ctrl-stk-size.

THEOREM: proper-p-statep-lr->p-lessp-max-temp-stk-size
proper-p-statep (lr->p (l))
→ ((p-max-temp-stk-size (l) < exp (2, p-word-size (l))) = **t**)

EVENT: Disable proper-p-statep-lr->p-lessp-max-temp-stk-size.

THEOREM: proper-p-statep-lr->p-equal-word-size-0
proper-p-statep (lr->p (l)) → (p-word-size (l) ≠ 0)

EVENT: Disable proper-p-statep-lr->p-equal-word-size-0.

THEOREM: proper-p-ctrl-stkp-lr-apply-subr
(proper-p-statep (lr->p (*new-l*))
∧ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
∧ lr-programs-properp (l, *table*)
∧ listp (lr-expr (l))
∧ subrp (car (lr-expr (l)))
∧ (car (lr-expr (l)) ≠ 'if)
∧ (p-prog-segment (l) = p-prog-segment (*new-l*))
∧ (area-name (p-pc (l)) = area-name (p-pc (*new-l*)))
∧ (p-psw (lr-apply-subr (l, *new-l*)) = 'run))
→ proper-p-ctrl-stkp (cdr (p-ctrl-stk (*new-l*)),
area-name (ret-pc (car (p-ctrl-stk (*new-l*)))),
lr->p (lr-apply-subr (l, *new-l*)))

THEOREM: proper-p-prog-segmentp-lr-apply-subr
(proper-p-statep (lr->p (*new-l*))

$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ listp (lr-expr (l))
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (p-prog-segment (l) = p-prog-segment (*new-l*))
$\wedge$ (area-name (p-pc (l)) = area-name (p-pc (*new-l*)))
$\wedge$ (p-psw (lr-apply-subr (l, *new-l*)) = 'run)
$\wedge$ (*progs* = p-prog-segment (*new-l*)))
$\rightarrow$ proper-p-prog-segmentp (comp-programs (*progs*),
lr->p (lr-apply-subr (l, *new-l*)))

THEOREM: proper-p-state-p-p-run-subr-opener-1
(proper-p-statep (p) $\wedge$ (p-psw (p-run-subr (*subr*, p)) = 'run))
$\rightarrow$ proper-p-temp-stkp (p-temp-stk (p-run-subr (*subr*, p)), p-run-subr (*subr*, p))

THEOREM: proper-p-temp-stkp-lr-apply-subr
(proper-p-statep (lr->p (*new-l*))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ listp (lr-expr (l))
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (p-prog-segment (l) = p-prog-segment (*new-l*))
$\wedge$ (area-name (p-pc (l)) = area-name (p-pc (*new-l*)))
$\wedge$ (p-psw (lr-apply-subr (l, *new-l*)) = 'run))
$\rightarrow$ proper-p-temp-stkp (p-temp-stk (lr-apply-subr (l, *new-l*)),
lr->p (lr-apply-subr (l, *new-l*)))

THEOREM: proper-p-state-p-p-run-subr-opener-2
(proper-p-statep (p) $\wedge$ (p-psw (p-run-subr (*subr*, p)) = 'run))
$\rightarrow$ (p-max-temp-stk-size (p) $\not<$ length (p-temp-stk (p-run-subr (*subr*, p))))

THEOREM: not-lessp-length-p-temp-stk-lr-apply-subr
(proper-p-statep (lr->p (*new-l*))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ listp (lr-expr (l))
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (p-prog-segment (l) = p-prog-segment (*new-l*))
$\wedge$ (area-name (p-pc (l)) = area-name (p-pc (*new-l*)))
$\wedge$ (p-psw (lr-apply-subr (l, *new-l*)) = 'run)
$\wedge$ (p-max-temp-stk-size (l) = p-max-temp-stk-size (*new-l*)))
$\rightarrow$ (p-max-temp-stk-size (l)
$\not<$ length (p-temp-stk (lr-apply-subr (l, *new-l*)))))

THEOREM: proper-p-state-p-p-run-subr-opener-3
(proper-p-statep (*p*) ∧ (p-psw (p-run-subr (*subr*, *p*)) = 'run))
→ proper-p-data-segmentp (p-data-segment (p-run-subr (*subr*, *p*)),
p-run-subr (*subr*, *p*))

THEOREM: proper-p-data-segmentp-lr-apply-subr
(proper-p-statep (lr->p (*new-l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ listp (lr-expr (*l*))
∧ subrp (car (lr-expr (*l*)))
∧ (car (lr-expr (*l*)) ≠ 'if)
∧ (p-prog-segment (*l*) = p-prog-segment (*new-l*))
∧ (area-name (p-pc (*l*)) = area-name (p-pc (*new-l*)))
∧ (p-psw (lr-apply-subr (*l*, *new-l*)) = 'run))
→ proper-p-data-segmentp (p-data-segment (lr-apply-subr (*l*, *new-l*)),
lr->p (lr-apply-subr (*l*, *new-l*)))

THEOREM: lr-programs-properp-lr-set-pos
lr-programs-properp (lr-set-pos (*l*, *pos*), *table*)
= lr-programs-properp (*l*, *table*)

THEOREM: proper-p-statep-lr-apply-subr
(listp (lr-expr (*l*))
∧ subrp (car (lr-expr (*l*)))
∧ (car (lr-expr (*l*)) ≠ 'if)
∧ proper-p-statep (lr->p (*l*))
∧ proper-p-statep (lr->p (lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)) = 'run)
∧ (p-psw (lr-apply-subr (*l*, lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)))
= 'run))
→ proper-p-statep (lr->p (lr-apply-subr (*l*,
lr-eval ('list,
lr-set-pos (*l*, *pos*),
c))))

THEOREM: cdr-untag-lr-p-pc-lr-funcall
(p-psw (lr-funcall (*l*, *new-l*)) = 'run)
→ (cdr (untag (lr-p-pc (lr-funcall (*l*, *new-l*))))) = 0)

THEOREM: listp-p-ctrl-stk-lr-funcall
listp (p-ctrl-stk (*new-l*)) → listp (p-ctrl-stk (lr-funcall (*l*, *new-l*)))

THEOREM: proper-p-framep-top-p-ctrl-stk-lr-funcall
(p-psw (lr-funcall (l, *new-l*)) = 'run)
→ (listp (car (p-ctrl-stk (lr-funcall (l, *new-l*))))
∧ listp (cdr (car (p-ctrl-stk (lr-funcall (l, *new-l*)))))
∧ (cddr (car (p-ctrl-stk (lr-funcall (l, *new-l*))))) = **nil**)
∧ (ret-pc (car (p-ctrl-stk (lr-funcall (l, *new-l*)))))
= add-addr (lr-return-pc (l), 1)))

THEOREM: car-untag-p-pc-lr-funcall
(p-psw (lr-funcall (l, *new-l*)) = 'run)
→ (car (untag (p-pc (lr-funcall (l, *new-l*)))))
= user-fname (car (lr-expr (l))))

THEOREM: area-name-p-pc-lr-funcall
(p-psw (lr-funcall (l, *new-l*)) = 'run)
→ (area-name (p-pc (lr-funcall (l, *new-l*))) = user-fname (car (lr-expr (l))))

THEOREM: strip-cars-bindings-top-p-ctrl-stk-lr-funcall
(p-psw (lr-funcall (l, *new-l*)) = 'run)
→ (strip-cars (bindings (car (p-ctrl-stk (lr-funcall (l, *new-l*))))))
= append (formal-vars (assoc (user-fname (car (lr-expr (l))),
p-prog-segment (l))),
strip-cars (temp-var-dcls (assoc (user-fname (car (lr-expr (l))),
p-prog-segment (l))))))

THEOREM: formal-vars-assoc-comp-programs-lr-programs-properp
(definedp (*name*, cdr (p-prog-segment (l))) ∧ lr-programs-properp (l, *table*))
→ (formal-vars (assoc (*name*, comp-programs (p-prog-segment (l)))))
= formal-vars (assoc (*name*, p-prog-segment (l))))

THEOREM: temp-var-dcls-assoc-comp-programs-lr-programs-properp
(definedp (*name*, cdr (p-prog-segment (l))) ∧ lr-programs-properp (l, *table*))
→ (temp-var-dcls (assoc (*name*, comp-programs (p-prog-segment (l)))))
= temp-var-dcls (assoc (*name*, p-prog-segment (l))))

THEOREM: definedp-comp-programs-definedp-lr-programs-properp
(definedp (*name*, cdr (p-prog-segment (l))) ∧ lr-programs-properp (l, *table*))
→ definedp (*name*, comp-programs (p-prog-segment (l)))

THEOREM: definedp-lr-funcall-prog-segment
(listp (lr-expr (l))
∧ (¬ subrp (car (lr-expr (l))))
∧ (car (lr-expr (l)) ≠ 'quote)
∧ litatom (car (lr-expr (l)))
∧ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))

$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (*progs* = cdr (p-prog-segment (l)))
$\rightarrow$ definedp (user-fname (car (lr-expr (l))), *progs*)

THEOREM: pop-p-ctrl-stk-lr-funcall
(p-psw (lr-funcall (l, *new-l*)) = `'run`)
$\rightarrow$ (cdr (p-ctrl-stk (lr-funcall (l, *new-l*))) = p-ctrl-stk (*new-l*))

THEOREM: proper-p-alistp-lr-funcall
(lr-programs-properp (l, *table*)
$\wedge$ definedp (user-fname (car (lr-expr (l))), cdr (p-prog-segment (*new-l*)))
$\wedge$ proper-p-prog-segmentp (comp-programs (p-prog-segment (*new-l*)),
lr->p (*new-l*))
$\wedge$ proper-p-temp-stkp (p-temp-stk (*new-l*), lr->p (*new-l*))
$\wedge$ (p-psw (lr-funcall (l, *new-l*)) = `'run`)
$\wedge$ (p-prog-segment (l) = p-prog-segment (*new-l*)))
$\rightarrow$ proper-p-alistp (bindings (car (p-ctrl-stk (lr-funcall (l, *new-l*)))),
lr->p (lr-funcall (l, *new-l*)))

THEOREM: proper-p-ctrl-stkp-lr-funcall
(proper-p-ctrl-stkp (cdr (*ctrl-stk*),
area-name (ret-pc (car (*ctrl-stk*))),
lr->p (*new-l*))
$\wedge$ proper-p-framep (top (*ctrl-stk*), *name*, lr->p (*new-l*))
$\wedge$ listp (*ctrl-stk*))
$\rightarrow$ proper-p-ctrl-stkp (*ctrl-stk*, *name*, lr->p (lr-funcall (l, *new-l*)))

THEOREM: not-lessp-p-max-ctrl-stk-size-lr-funcall
(listp (lr-expr (l))
$\wedge$ ($\neg$ subrp (car (lr-expr (l))))
$\wedge$ (car (lr-expr (l)) $\neq$ `'quote`)
$\wedge$ litatom (car (lr-expr (l)))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-funcall (l, *new-l*)) = `'run`)
$\wedge$ (p-max-ctrl-stk-size (l) = p-max-ctrl-stk-size (*new-l*))
$\wedge$ (p-prog-segment (l) = p-prog-segment (*new-l*)))
$\rightarrow$ (p-max-ctrl-stk-size (l)
$\not<$ p-ctrl-stk-size (p-ctrl-stk (lr-funcall (l, *new-l*)))))

THEOREM: offset-p-pc-lr-funcall
(p-psw (lr-funcall (l, *new-l*)) = `'run`)
$\rightarrow$ (offset (p-pc (lr-funcall (l, *new-l*))) = **nil**)

THEOREM: lr-eval-t-lr-funcall-p-psw-run
(p-psw (lr-eval (**t**, lr-funcall (l, *new-l*), c)) = `'run`)
$\rightarrow$ (p-psw (lr-funcall (l, *new-l*)) = `'run`)

THEOREM: proper-p-temp-stkp-lr-funcall
(listp (lr-expr (l))
 $\wedge$ ($\neg$ subrp (car (lr-expr (l))))
 $\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
 $\wedge$ litatom (car (lr-expr (l)))
 $\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 $\wedge$ lr-programs-properp (l, *table*)
 $\wedge$ (p-psw (lr-funcall (l, *new-l*)) = 'run)
 $\wedge$ proper-p-temp-stkp (p-temp-stk (*new-l*), lr->p (*new-l*))
 $\wedge$ (p-prog-segment (l) = p-prog-segment (*new-l*)))
$\rightarrow$ proper-p-temp-stkp (p-temp-stk (lr-funcall (l, *new-l*)),
lr->p (lr-funcall (l, *new-l*)))

THEOREM: popn-nlistp
($\neg$ listp (x)) $\rightarrow$ ($\neg$ listp (popn (n, x)))

THEOREM: length-popn-lessp-fact
length (*list*) $\not<$ length (popn (n, *list*))

EVENT: Disable popn-nlistp.

THEOREM: not-lessp-p-max-temp-stk-size-lr-funcall
((p-max-temp-stk-size (l) $\not<$ length (p-temp-stk (*new-l*)))
 $\wedge$ (p-max-temp-stk-size (l) = p-max-temp-stk-size (*new-l*)))
$\rightarrow$ (p-max-temp-stk-size (l) $\not<$ length (p-temp-stk (lr-funcall (l, *new-l*))))

THEOREM: listp-label-instrs
listp (label-instrs (*list*, n)) = listp (*list*)

THEOREM: listp-comp-body
listp (comp-body (*body*))

THEOREM: lessp-offset-lr-return-pc-lr-p-c-size-good-posp
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 $\wedge$ lr-programs-properp (l, *table*)
 $\wedge$ (subrp (car (lr-expr (l))) $\vee$ litatom (car (lr-expr (l))))
 $\wedge$ (car (lr-expr (l)) $\neq$ 'if)
 $\wedge$ (car (lr-expr (l)) $\neq$ 'quote))
$\rightarrow$ ((1 + offset (lr-return-pc (l)))
$<$ length (program-body (assoc (area-name (p-pc (l)),
comp-programs (p-prog-segment (l))))))

THEOREM: proper-p-statep-lr-funcall
(proper-p-statep (lr->p (lr-eval ('list, lr-set-pos (l, *pos*), c)))
 $\wedge$ listp (lr-expr (l))

$\wedge$ ($\neg$ subrp (car (lr-expr (l))))
$\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ litatom (car (lr-expr (l)))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval ('list, lr-set-pos (l, *pos*), c)) = 'run)
$\wedge$ (p-psw (lr-funcall (l, lr-eval ('list, lr-set-pos (l, *pos*), c))) = 'run))
$\rightarrow$ proper-p-statep (lr->p (lr-funcall (l,
lr-eval ('list, lr-set-pos (l, *pos*), c))))

THEOREM: proper-p-statep-lr-set-expr-lr-pop-cstk
let *l2* **be** lr-eval (**t**, lr-funcall (l, *new-l*), $c - 1$)
in
(definedp (area-name (p-pc (l)), p-prog-segment (l))
$\wedge$ (cdr (p-ctrl-stk (*l2*)) = p-ctrl-stk (*new-l*))
$\wedge$ (cdr (p-ctrl-stk (*new-l*)) = cdr (p-ctrl-stk (l)))
$\wedge$ (strip-cars (bindings (car (p-ctrl-stk (*new-l*))))
= strip-cars (bindings (car (p-ctrl-stk (l)))))
$\wedge$ proper-p-statep (lr->p (*l2*))
$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ (p-psw (*l2*) = 'run)
$\wedge$ same-signature (p-data-segment (*new-l*), p-data-segment (*l2*))
$\wedge$ (p-prog-segment (*new-l*) = p-prog-segment (l))
$\wedge$ (area-name (p-pc (*new-l*)) = area-name (p-pc (l)))
$\wedge$ (*pos* = offset (p-pc (l))))
$\rightarrow$ proper-p-statep (lr->p (lr-set-expr (lr-pop-cstk (*l2*), l, *pos*))) **endlet**

THEOREM: p-psw-lr-eval-flag-list-flag-t
((p-psw (lr-eval ('list, lr-set-expr (lr-eval (**t**, l, c), l, *pos*), c)) = 'run)
$\wedge$ listp (offset (p-pc (l)))
$\wedge$ listp (lr-expr-list (l)))
$\rightarrow$ (p-psw (lr-eval (**t**, l, c)) = 'run)

THEOREM: lr-programs-properp-lr-set-expr
lr-programs-properp (lr-set-expr (*l1*, *l2*, *pos*), *table*)
= lr-programs-properp (*l2*, *table*)

THEOREM: lr-programs-properp-lr-pop-tstk
lr-programs-properp (lr-pop-tstk (l), *table*) = lr-programs-properp (l, *table*)

THEOREM: lr-programs-properp-lr-funcall
(listp (lr-expr (l))
$\wedge$ ($\neg$ subrp (car (lr-expr (l))))
$\wedge$ (car (lr-expr (l)) $\neq$ 'quote)

$\wedge$ litatom (car (lr-expr (l)))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*))
$\rightarrow$ lr-programs-properp (lr-funcall (l, lr-eval (`'list`, lr-set-pos (l, *pos*), c)), *table*)

THEOREM: proper-p-statep-lr->p-lr-set-expr
(lr-programs-properp ($l2$, *table*)
$\wedge$ lr-programs-properp ($l1$, *table*)
$\wedge$ proper-p-statep (lr->p ($l2$))
$\wedge$ proper-p-statep (lr->p ($l1$))
$\wedge$ (cdr (p-ctrl-stk ($l1$)) = cdr (p-ctrl-stk ($l2$)))
$\wedge$ (strip-cars (bindings (car (p-ctrl-stk ($l1$))))
= strip-cars (bindings (car (p-ctrl-stk ($l2$)))))
$\wedge$ (p-prog-segment ($l1$) = p-prog-segment ($l2$))
$\wedge$ (p-word-size ($l1$) = p-word-size ($l2$))
$\wedge$ (p-max-ctrl-stk-size ($l1$) = p-max-ctrl-stk-size ($l2$))
$\wedge$ (p-max-temp-stk-size ($l1$) = p-max-temp-stk-size ($l2$)))
$\rightarrow$ proper-p-statep (lr->p (lr-set-expr ($l1$, $l2$, *pos*)))

THEOREM: lr-programs-properp-lr-if-ok
lr-programs-properp (lr-if-ok (l), *table*) = lr-programs-properp (l, *table*)

THEOREM: proper-p-statep-lr-if-ok
proper-p-statep (lr->p (lr-if-ok (l))) = proper-p-statep (lr->p (l))

THEOREM: lr-eval-preserves-proper-p-statep-lr->p
(proper-p-statep (lr->p (l))
$\wedge$ good-posp (*flag*, offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval (*flag*, l, c)) = `'run`))
$\rightarrow$ (proper-p-statep (lr->p (lr-eval (*flag*, l, c)))
$\wedge$ (strip-cars (bindings (car (p-ctrl-stk (lr-eval (*flag*, l, c))))))
= strip-cars (bindings (car (p-ctrl-stk (l)))))
$\wedge$ (cdr (p-ctrl-stk (lr-eval (*flag*, l, c))) = cdr (p-ctrl-stk (l)))
$\wedge$ ((*flag* = `'list`) $\vee$ listp (p-temp-stk (lr-eval (*flag*, l, c))))
$\wedge$ same-signature (p-data-segment (l), p-data-segment (lr-eval (*flag*, l, c))))

THEOREM: lr-params-lr-eval
lr-params (*frame*, lr-eval (*flag*, l, c)) = lr-params (*frame*, l)

THEOREM: lr-temps-lr-eval
lr-temps (*frame*, lr-eval (*flag*, l, c)) = lr-temps (*frame*, l)

```
;; Later LR-FREE-LIST-NODES will filter out those nodes that are
;; part of for example pack's or user-defined types that are larger than one
;; node (i.e. have more than two accessors).
```

DEFINITION:
lr-free-list-nodes (*addr*, *data-seg*)
= **if** offset (*addr*) < LR-NODE-SIZE **then nil**
else let *sub-addr* **be** sub-addr (*addr*, LR-NODE-SIZE)
in
if type (fetch (add-addr (*sub-addr*, LR-REF-COUNT-OFFSET),
data-seg))
= 'addr
then cons (*sub-addr*,
lr-free-list-nodes (*sub-addr*, *data-seg*))
else lr-free-list-nodes (*sub-addr*, *data-seg*) **endif endlet endif**

THEOREM: length-delete-member
(*addr* ∈ *node-list*)
→ (length (delete (*addr*, *node-list*)) = (length (*node-list*) − 1))

```
;; Returns smallest address such that the address is too large to be
;; a pointer to a node in DATA-SEG.
```

DEFINITION:
lr-max-node (*data-seg*)
= tag ('addr,
cons (LR-HEAP-NAME, length (value (LR-HEAP-NAME, *data-seg*)) − 1))

DEFINITION:
lr-check-free-nodes (*addr*, *node-list*, *data-seg*, *max-addr*)
= **if** *addr* ∈ *node-list*
then lr-check-free-nodes (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET),
data-seg),
delete (*addr*, *node-list*),
data-seg,
max-addr)
else *addr* = *max-addr* **endif**

DEFINITION:
lr-proper-free-listp (*data-seg*)
= (adpp (untag (LR-FP-ADDR), *data-seg*)
∧ lr-check-free-nodes (lr-fetch-fp (*data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg,
lr-max-node (*data-seg*)))

EVENT: Disable lr-proper-free-listp.

DEFINITION:
lr-check-f-addrp (*addr*, *data-seg*) = (*addr* = LR-F-ADDR)

EVENT: Disable lr-check-f-addrp.

DEFINITION:
lr-check-undef-addrp (*addr*, *data-seg*) = (*addr* = LR-UNDEF-ADDR)

EVENT: Disable lr-check-undef-addrp.

DEFINITION:
lr-check-numberp-addrp (*addr*, *data-seg*)
= ((type (fetch (add-addr (*addr*, LR-UNBOX-NAT-OFFSET), *data-seg*)) = 'nat)
∧ lr-good-pointerp (fetch (add-addr (*addr*, 3), *data-seg*), *data-seg*)
∧ (untag (fetch (add-addr (*addr*, LR-UNBOX-NAT-OFFSET), *data-seg*)) ∈ **N**))

EVENT: Disable lr-check-numberp-addrp.

DEFINITION:
lr-check-listp-addrp (*addr*, *data-seg*)
= (lr-good-pointerp (fetch (add-addr (*addr*, LR-CAR-OFFSET), *data-seg*),
data-seg)
∧ lr-good-pointerp (fetch (add-addr (*addr*, LR-CDR-OFFSET), *data-seg*),
data-seg))

EVENT: Disable lr-check-listp-addrp.

DEFINITION:
lr-proper-heapp-nodep (*addr*, *data-seg*)
= **if** ¬ lr-nodep (*addr*, *data-seg*) **then f**
elseif type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*))
= 'addr
then offset (*addr*) ≮ (LR-NODE-SIZE + offset (LR-F-ADDR))
elseif type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*))
≠ 'nat **then f**
elseif type (fetch (*addr*, *data-seg*)) ≠ 'nat **then f**
elseif untag (fetch (*addr*, *data-seg*)) = LR-UNDEFINED-TAG
then lr-check-undef-addrp (*addr*, *data-seg*)
elseif untag (fetch (*addr*, *data-seg*)) = LR-FALSE-TAG
then lr-check-f-addrp (*addr*, *data-seg*)

elseif offset (*addr*) < offset (LR-T-ADDR) **then f**
elseif untag (fetch (*addr*, *data-seg*)) = LR-TRUE-TAG **then t**
elseif untag (fetch (*addr*, *data-seg*)) = LR-ADD1-TAG
then lr-check-numberp-addrp (*addr*, *data-seg*)
elseif untag (fetch (*addr*, *data-seg*)) = LR-CONS-TAG
then lr-check-listp-addrp (*addr*, *data-seg*)
else f endif

EVENT: Disable lr-proper-heapp-nodep.

DEFINITION:
lr-proper-heapp2 (*addr*, *data-seg*)
= **if** offset (*addr*) < LR-NODE-SIZE **then t**
else let *sub-addr* **be** sub-addr (*addr*, LR-NODE-SIZE)
in
lr-proper-heapp-nodep (*sub-addr*, *data-seg*)
∧ lr-proper-heapp2 (*sub-addr*, *data-seg*) **endlet endif**

DEFINITION:
lr-valp (*value*, *addr*, *data-seg*)
= **if** lr-good-pointerp (*addr*, *data-seg*)
then let *tag* **be** untag (fetch (*addr*, *data-seg*))
in
if listp (*value*)
then (*tag* = LR-CONS-TAG)
∧ lr-valp (car (*value*),
fetch (add-addr (*addr*, LR-CAR-OFFSET),
data-seg),
data-seg)
∧ lr-valp (cdr (*value*),
fetch (add-addr (*addr*, LR-CDR-OFFSET),
data-seg),
data-seg)
elseif truep (*value*) **then** *tag* = LR-TRUE-TAG
elseif falsep (*value*) **then** *tag* = LR-FALSE-TAG
elseif *value* ∈ **N**
then (*tag* = LR-ADD1-TAG)
∧ (*value* = untag (fetch (add-addr (*addr*,
LR-UNBOX-NAT-OFFSET),
data-seg)))
else f endif endlet
else f endif

DEFINITION:

lr-proper-heapp1 (*addr*, *data-seg*)
= (lr-proper-heapp2 (*addr*, *data-seg*)
  ∧ lr-valp (**t**, LR-T-ADDR, *data-seg*)
  ∧ lr-valp (0, LR-0-ADDR, *data-seg*))

EVENT: Disable lr-proper-heapp1.

```
;; This is the minimum heap that allows all the predefineds to be defined.
```

DEFINITION:
lr-minimum-heapp (*data-seg*)
= (adpp (untag (LR-UNDEF-ADDR), *data-seg*)
  ∧ adpp (untag (LR-F-ADDR), *data-seg*)
  ∧ adpp (untag (LR-T-ADDR), *data-seg*)
  ∧ adpp (untag (LR-0-ADDR), *data-seg*)
  ∧ adpp (untag (add-addr (LR-0-ADDR, LR-NODE-SIZE)), *data-seg*))

EVENT: Disable lr-minimum-heapp.

```
;; This needs to be augmented to test that the word-size is big enough to
;; hold piton tags.
```

DEFINITION:
lr-proper-heapp (*data-seg*)
= (lr-minimum-heapp (*data-seg*)
  ∧ lr-nodep (lr-max-node (*data-seg*), *data-seg*)
  ∧ lr-proper-free-listp (*data-seg*)
  ∧ lr-proper-heapp1 (lr-max-node (*data-seg*), *data-seg*))

EVENT: Disable lr-proper-heapp.

DEFINITION:
lr-check-result1 (*value*, *temp-stk*, *data-seg*)
= **if** listp (*value*)
  then lr-valp (car (*value*), top (*temp-stk*), *data-seg*)
    ∧ lr-check-result1 (cdr (*value*), pop (*temp-stk*), *data-seg*)
  else t endif

DEFINITION:
lr-check-result (*flag*, *value*, *temp-stk*, *data-seg*, *orig-temp-stk*)
= ((*orig-temp-stk* = **if** *flag* = 'list
    then restn (length (*value*), *temp-stk*)
    else cdr (*temp-stk*) **endif**)
  ∧ **if** *flag* = 'list

then lr-check-result1 (reverse (*value*), *temp-stk*, *data-seg*)
else lr-valp (*value*, top (*temp-stk*), *data-seg*) **endif**
$\wedge$ lr-proper-heapp (*data-seg*))

EVENT: Disable lr-check-result.

DEFINITION:
lr-s-similar-params (*s-params*, *lr-params*, *data-seg*)
= **if** listp (*s-params*)
then if listp (*lr-params*)
then (caar (*s-params*) = caar (*lr-params*))
$\wedge$ lr-valp (cdar (*s-params*), cdar (*lr-params*), *data-seg*)
$\wedge$ lr-s-similar-params (cdr (*s-params*),
cdr (*lr-params*),
data-seg)
else f endif
else *lr-params* $\simeq$ **nil endif**

DEFINITION:
lr-s-similar-temps (*s-temps*, *lr-temps*, *data-seg*)
= **if** listp (*s-temps*)
then if listp (*lr-temps*)
then if cdar (*lr-temps*) = LR-UNDEF-ADDR **then** $\neg$ cadar (*s-temps*)
else cadar (*s-temps*)
$\wedge$ lr-valp (caddar (*s-temps*),
cdar (*lr-temps*),
data-seg) **endif**
$\wedge$ lr-s-similar-temps (cdr (*s-temps*),
cdr (*lr-temps*),
data-seg)
else f endif
else *lr-temps* $\simeq$ **nil endif**

DEFINITION:
lr-s-similar-const-table (*table*, *data-seg*)
= **if** listp (*table*)
then lr-valp (caar (*table*), cdar (*table*), *data-seg*)
$\wedge$ lr-s-similar-const-table (cdr (*table*), *data-seg*)
else t endif

DEFINITION:
lr-s-similar-statesp (*s-params*, *s-temps*, *l*, *table*)
= (lr-s-similar-params (*s-params*,
lr-params (top (p-ctrl-stk (*l*)), *l*),

p-data-segment (*l*))
∧ lr-s-similar-temps (*s-temps*,
lr-temps (top (p-ctrl-stk (*l*)), *l*),
p-data-segment (*l*))
∧ lr-s-similar-const-table (*table*, p-data-segment (*l*)))

EVENT: Disable lr-s-similar-statesp.

THEOREM: p-accessors-s->lr1
(p-pc (s->lr1 (*s*, *l*, *table*)) = tag ('pc, cons (s-pname (*s*), s-pos (*s*))))
∧ (p-ctrl-stk (s->lr1 (*s*, *l*, *table*)) = p-ctrl-stk (*l*))
∧ (p-temp-stk (s->lr1 (*s*, *l*, *table*)) = p-temp-stk (*l*))
∧ (p-prog-segment (s->lr1 (*s*, *l*, *table*))
= lr-compile-programs (s-progs (*s*), *table*))
∧ (p-data-segment (s->lr1 (*s*, *l*, *table*)) = p-data-segment (*l*))
∧ (p-max-ctrl-stk-size (s->lr1 (*s*, *l*, *table*)) = p-max-ctrl-stk-size (*l*))
∧ (p-max-temp-stk-size (s->lr1 (*s*, *l*, *table*)) = p-max-temp-stk-size (*l*))
∧ (p-word-size (s->lr1 (*s*, *l*, *table*)) = p-word-size (*l*))
∧ (p-psw (s->lr1 (*s*, *l*, *table*)) = s-err-flag (*s*))

THEOREM: s-eval-err-flag-not-run-fact
(s-err-flag (*s*) ≠ 'run) → (s-eval (*flag*, *s*, *clock*) = *s*)

;; OFFSET

THEOREM: offset-tag-cons
offset (tag (*tag*, cons (*area*, *offset*))) = *offset*

;; ADP-NAME

THEOREM: adp-name-cons
adp-name (cons (*x*, *y*)) = *x*

;; OFFSET-SUB-ADDR -- see above

;; LR-PROPER-P-AREASP

THEOREM: definedp-litatom-lr-proper-p-areas
((¬ litatom (*name*)) ∧ lr-proper-p-areasp (*data-seg*))
→ (¬ definedp (*name*, *data-seg*))

EVENT: Disable definedp-litatom-lr-proper-p-areas.

THEOREM: member-lr-free-list-nodes-type-addr
(type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) ≠ 'addr)
→ (*addr* ∉ lr-free-list-nodes (*max-addr*, *data-seg*))

EVENT: Disable member-lr-free-list-nodes-type-addr.

THEOREM: lessp-length-deposit
length (cdr (assoc (*name*, deposit (*any*, *addr*, *data-seg*))))
≮ length (cdr (assoc (*name*, *data-seg*)))

;; GET

THEOREM: definedp-listp-cdr-assoc-lr-proper-p-areasp
lr-proper-p-areasp (*data-seg*)
→ (listp (cdr (assoc (*area-name*, *data-seg*)))
= definedp (*area-name*, *data-seg*))

EVENT: Disable definedp-listp-cdr-assoc-lr-proper-p-areasp.

;; LR-MINIMUM-HEAPP

THEOREM: lr-minimum-heapp-opener-adpp-lr-f-addr
lr-minimum-heapp (*data-seg*) → adpp (identity (untag (LR-F-ADDR)), *data-seg*)

THEOREM: lr-minimum-heapp-opener-adpp-lr-t-addr
lr-minimum-heapp (*data-seg*) → adpp (identity (untag (LR-T-ADDR)), *data-seg*)

THEOREM: lr-minimum-heapp-opener-adpp-lr-0-addr
lr-minimum-heapp (*data-seg*) → adpp (identity (untag (LR-0-ADDR)), *data-seg*)

THEOREM: lr-minimum-heapp-opener-adpp-lr-undef-addr
lr-minimum-heapp (*data-seg*)
→ adpp (identity (untag (LR-UNDEF-ADDR)), *data-seg*)

THEOREM: lr-boundary-offsetp-sub1-length-heap-name
lr-boundary-nodep (lr-max-node (*data-seg*))
→ lr-boundary-offsetp (length (cdr (assoc (identity (LR-HEAP-NAME), *data-seg*))) − 1)

THEOREM: lessp-lr-boundary-offsetp-nodep-plus-node-size-fact-2
(lr-boundary-offsetp (*offset1*)
∧ lr-boundary-offsetp (*offset2*)
∧ (*n* < LR-NODE-SIZE))
→ (((*n* + *offset1*) < *offset2*) = (*offset1* < *offset2*))

THEOREM: lr-boundary-offsetp-times-lr-node-size-anything
lr-boundary-offsetp (identity (LR-NODE-SIZE) ∗ *x*)

THEOREM: lr-boundary-offsetp-difference-not-equal-lessp-fact-2
(lr-boundary-offsetp (*x*) ∧ lr-boundary-offsetp (*y*) ∧ (*x* ∈ **N**) ∧ (*x* < *y*))
→ (((*y* − LR-NODE-SIZE) < *x*) = **f**)

THEOREM: lr-minimum-heapp-opener-2
lr-minimum-heapp (*data-seg*)
→ (identity (LR-MINIMUM-HEAP-SIZE)
< length (cdr (assoc (identity (LR-HEAP-NAME), *data-seg*))))

EVENT: Disable lr-minimum-heapp-opener-2.

THEOREM: lr-minimum-heapp-opener-3
lr-minimum-heapp (*data-seg*) → definedp (identity (LR-HEAP-NAME), *data-seg*)

EVENT: Disable lr-minimum-heapp-opener-3.

`;; LR-PROPER-FREE-LISTP`

DEFINITION:
lr-node-listp (*list*, *data-seg*)
= **if** listp (*list*)
then lr-nodep (car (*list*), *data-seg*)
∧ lr-node-listp (cdr (*list*), *data-seg*)
else t endif

EVENT: Disable lr-node-listp.

THEOREM: adpp-adpp-sub-addr
adpp (untag (*addr*), *data-seg*) → adpp (untag (sub-addr (*addr*, *n*)), *data-seg*)

THEOREM: lr-node-listp-lr-free-list-nodes
(lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = LR-HEAP-NAME)
∧ adpp (untag (*addr*), *data-seg2*)
∧ (type (*addr*) = `'addr`))
→ lr-node-listp (lr-free-list-nodes (*addr*, *data-seg1*), *data-seg2*)

THEOREM: lr-nodep-member-lr-node-listp
(lr-node-listp (*list*, *data-seg*) ∧ (*node* ∈ *list*))
→ ((type (*node*) = `'addr`)
∧ (cddr (*node*) = **nil**)
∧ listp (*node*)
∧ adpp (untag (*node*), *data-seg*)
∧ lr-boundary-nodep (*node*)
∧ (area-name (*node*) = LR-HEAP-NAME))

EVENT: Disable lr-nodep-member-lr-node-listp.

THEOREM: lr-max-node-lr-nodep-opener-facts
(type (lr-max-node (*data-seg*)) = 'addr)
∧ (cddr (lr-max-node (*data-seg*)) = **nil**)
∧ (area-name (lr-max-node (*data-seg*)) = LR-HEAP-NAME)

THEOREM: lr-max-node-adpp-definedp-lr-heap-name
lr-proper-p-areasp (*data-seg*)
→ (adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
= definedp (LR-HEAP-NAME, *data-seg*))

THEOREM: offset-lr-max-node
offset (lr-max-node (*data-seg*))
= (length (cdr (assoc (identity (LR-HEAP-NAME), *data-seg*))) − 1)

EVENT: Disable lr-max-node.

THEOREM: lr-proper-free-listp-opener-1
lr-proper-free-listp (*data-seg*)
→ adpp (identity (untag (LR-FP-ADDR)), *data-seg*)

THEOREM: lr-proper-free-listp-opener-2
(lr-proper-free-listp (*data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ ((type (fetch (identity (LR-FP-ADDR), *data-seg*)) = 'addr)
∧ (cddr (fetch (identity (LR-FP-ADDR), *data-seg*)) = **nil**)
∧ listp (fetch (identity (LR-FP-ADDR), *data-seg*))
∧ adpp (untag (fetch (identity (LR-FP-ADDR), *data-seg*)), *data-seg*)
∧ lr-boundary-nodep (fetch (identity (LR-FP-ADDR), *data-seg*))
∧ (area-name (fetch (identity (LR-FP-ADDR), *data-seg*))
= LR-HEAP-NAME))

THEOREM: lr-proper-free-listp-opener-2-adpp-untag-numberp-offset
(lr-proper-free-listp (*data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (offset (fetch (identity (LR-FP-ADDR), *data-seg*)) ∈ **N**)

THEOREM: lr-proper-free-listp-opener-2-adpp-untag-listp
(lr-proper-free-listp (*data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ listp (untag (fetch (identity (LR-FP-ADDR), *data-seg*)))

THEOREM: plus-times-fact-1
$(n \not\simeq \mathtt{0}) \rightarrow (((n + (n * w)) < (d * n)) = ((1 + w) < d))$

EVENT: Disable plus-times-fact-1.

THEOREM: lessp-difference-fact-1
$(((x \textbf{ mod } n) = \mathtt{0}) \wedge ((y \textbf{ mod } n) = \mathtt{0}) \wedge (x < y) \wedge (x \in \mathbf{N}))$
$\rightarrow \quad ((x < (y - n)) = (x \neq (y - n)))$

THEOREM: lessp-difference-lr-boundary-offsetp-fact-1
$((\mathit{offset} \in \mathbf{N})$
$\wedge$ lr-boundary-offsetp (offset)
$\wedge$ lr-boundary-offsetp (y)
$\wedge$ $(\mathit{offset} < y))$
$\rightarrow$ $((\mathit{offset} < (y -$ identity (LR-NODE-SIZE)))
$=$ $(\mathit{offset} \neq (y -$ identity (LR-NODE-SIZE))))

THEOREM: lessp-lr-node-on-boundaryp-node-size
(lr-boundary-nodep $(\mathit{addr}) \wedge$ (offset $(\mathit{addr}) \in \mathbf{N}$))
$\rightarrow$ ((offset (addr) < identity (LR-NODE-SIZE)) = (offset $(\mathit{addr}) = \mathtt{0}$))

THEOREM: lessp-difference-node-size-sub-addr
((offset (addr) < offset $(\mathit{max\text{-}addr})$)
$\wedge$ (area-name (addr) = area-name $(\mathit{max\text{-}addr})$)
$\wedge$ lr-boundary-nodep $(\mathit{max\text{-}addr})$
$\wedge$ (type $(\mathit{max\text{-}addr})$ = `'addr`)
$\wedge$ (offset $(\mathit{max\text{-}addr}) \in \mathbf{N}$)
$\wedge$ (cddr $(\mathit{max\text{-}addr}) =$ **nil**)
$\wedge$ lr-boundary-nodep (addr)
$\wedge$ (type (addr) = `'addr`)
$\wedge$ (offset $(\mathit{addr}) \in \mathbf{N}$)
$\wedge$ (cddr $(\mathit{addr}) =$ **nil**)
$\wedge$ listp (untag (addr)))
$\rightarrow$ ((offset (addr) < (offset $(\mathit{max\text{-}addr})$ − identity (LR-NODE-SIZE)))
$=$ (sub-addr $(\mathit{max\text{-}addr}$, identity (LR-NODE-SIZE)) $\neq \mathit{addr}$))

THEOREM: lr-nodep-lr-proper-heapp-nodep
(lr-proper-heapp2 $(\mathit{max\text{-}addr}, \mathit{data\text{-}seg})$
$\wedge$ (offset (addr) < offset $(\mathit{max\text{-}addr})$)
$\wedge$ lr-nodep $(\mathit{max\text{-}addr}, \mathit{data\text{-}seg})$
$\wedge$ lr-nodep $(\mathit{addr}, \mathit{data\text{-}seg})$)
$\rightarrow$ lr-proper-heapp-nodep $(\mathit{addr}, \mathit{data\text{-}seg})$

EVENT: Disable lessp-difference-node-size-sub-addr.

THEOREM: adpp-area-name-offset-same
(listp (untag (*addr1*))
 ∧ (offset (*addr1*) ∈ **N**)
 ∧ (cddr (*addr1*) = **nil**)
 ∧ listp (untag (*addr2*))
 ∧ (offset (*addr2*) ∈ **N**)
 ∧ (cddr (*addr2*) = **nil**)
 ∧ (type (*addr1*) = type (*addr2*)))
→ ((*addr1* = *addr2*)
 = ((offset (*addr1*) = offset (*addr2*))
 ∧ (area-name (*addr1*) = area-name (*addr2*))))

THEOREM: lr-proper-heapp-nodep-tag-cons
((untag (fetch (*addr*, *data-seg*)) = LR-CONS-TAG)
 ∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) = 'nat)
 ∧ lr-proper-heapp-nodep (*addr*, *data-seg*)
 ∧ ((*offset* = LR-CAR-OFFSET) ∨ (*offset* = LR-CDR-OFFSET)))
→ lr-good-pointerp (fetch (add-addr (*addr*, *offset*), *data-seg*), *data-seg*)

THEOREM: adpp-add-addr-fact-2
(adpp (untag (*addr1*), *data-seg*)
 ∧ adpp (untag (add-addr (*addr1*, *n*)), *data-seg*)
 ∧ adpp (untag (*addr2*), *data-seg*)
 ∧ (¬ adpp (untag (add-addr (*addr2*, *n*)), *data-seg*))
 ∧ (area-name (*addr1*) = area-name (*addr2*)))
→ (offset (*addr1*) < offset (*addr2*))

THEOREM: fetch-lr-nodep-add-addr
((¬ adpp (untag (add-addr (*addr*, *n*)), *data-seg*)) ∧ lr-nodep (*addr*, *data-seg*))
→ (fetch (add-addr (*addr*, *n*), *data-seg*) = 0)

EVENT: Disable fetch-lr-nodep-add-addr.

THEOREM: untag-addr-addr-tag
untag (add-addr (tag (*tag*, *adp*), *n*)) = cons (car (*adp*), cdr (*adp*) + *n*)

THEOREM: lr-good-pointerp-lessp-offset-max-heap-node
(adpp (untag (*addr*), *data-seg*)
 ∧ lr-boundary-nodep (*addr*)
 ∧ listp (*addr*)
 ∧ (cddr (*addr*) = **nil**)
 ∧ (type (*addr*) = 'addr)
 ∧ (area-name (*addr*) = 'heap)
 ∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) = 'nat)

$\wedge$ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*)))
$\rightarrow$ (offset (*addr*)
$<$ (length (cdr (assoc (identity (LR-HEAP-NAME), *data-seg*))) $-$ 1))

EVENT: Disable lr-good-pointerp-lessp-offset-max-heap-node.

THEOREM: lr-proper-heapp-opener-1
lr-proper-heapp (*data-seg*)
$\rightarrow$ (lr-minimum-heapp (*data-seg*) $\wedge$ lr-proper-free-listp (*data-seg*))

THEOREM: lr-proper-heapp-opener-3
((*addr* = lr-max-node (*data-seg*)) $\wedge$ lr-proper-heapp (*data-seg*))
$\rightarrow$ lr-proper-heapp2 (*addr*, *data-seg*)

THEOREM: deposit-free-ptr-preserves-lr-valp
(adpp (untag (LR-FP-ADDR), *data-seg*) $\wedge$ lr-valp (*value*, *addr*, *data-seg*))
$\rightarrow$ lr-valp (*value*, *addr*, deposit (*anything*, identity (LR-FP-ADDR), *data-seg*))

THEOREM: lr-proper-p-areasp-deposit-anything-anywhere
lr-proper-p-areasp (*data-seg*)
$\rightarrow$ lr-proper-p-areasp (deposit (*anything*, *addr*, *data-seg*))

THEOREM: lr-node-listp-delete
lr-node-listp (*list*, *data-seg*)
$\rightarrow$ lr-node-listp (delete (*anything*, *list*), *data-seg*)

EVENT: Disable lr-node-listp-delete.

THEOREM: lr-node-listp-deposit-anything-at-all
lr-node-listp (*addr*, *data-seg*)
$\rightarrow$ lr-node-listp (*addr*, deposit (*anything*, *addr2*, *data-seg*))

EVENT: Disable lr-node-listp-deposit-anything-at-all.

THEOREM: cdr-assoc-member-strip-cdrs
definedp (*name*, *list*) $\rightarrow$ (cdr (assoc (*name*, *list*)) $\in$ strip-cdrs (*list*))

EVENT: Disable cdr-assoc-member-strip-cdrs.

THEOREM: lr-set-error-lr->p
lr->p (lr-set-error (*p*, *flag*)) = lr-set-error (lr->p (*p*), *flag*)

THEOREM: lr-params-lr-set-expr
((area-name (p-pc (*l*)) = area-name (p-pc (*l2*)))
 ∧ (p-prog-segment (*l*) = p-prog-segment (*l2*)))
→ (lr-params (*frame*, lr-set-expr (*l*, *l2*, *pos*)) = lr-params (*frame*, *l*))

THEOREM: lr-temps-lr-set-expr
((area-name (p-pc (*l*)) = area-name (p-pc (*l2*)))
 ∧ (p-prog-segment (*l*) = p-prog-segment (*l2*)))
→ (lr-temps (*frame*, lr-set-expr (*l*, *l2*, *pos*)) = lr-temps (*frame*, *l*))

THEOREM: p-current-program-lr-push-tstk
p-current-program (lr-push-tstk (*l*, *any*)) = p-current-program (*l*)

THEOREM: p-current-program-lr-set-temp
p-current-program (lr-set-temp (*l*, *value*, *var*)) = p-current-program (*l*)

THEOREM: p-current-program-lr-pop-tstk
p-current-program (lr-pop-tstk (*l*)) = p-current-program (*l*)

THEOREM: p-current-program-lr-do-temp-fetch
p-current-program (lr-do-temp-fetch (*l*)) = p-current-program (*l*)

THEOREM: strip-cars-restn
strip-cars (restn (*n*, *list*)) = restn (*n*, strip-cars (*list*))

EVENT: Disable strip-cars-restn.

THEOREM: strip-cars-firstn
strip-cars (firstn (*n*, *list*)) = firstn (*n*, strip-cars (*list*))

EVENT: Disable strip-cars-firstn.

THEOREM: lr-params-lr-pop-tstk
lr-params (*frame*, lr-pop-tstk (*l*)) = lr-params (*frame*, *l*)

THEOREM: lr-temps-lr-pop-tstk
lr-temps (*frame*, lr-pop-tstk (*l*)) = lr-temps (*frame*, *l*)

THEOREM: lr-minimum-heapp-same-signature
same-signature (*data-seg1*, *data-seg2*)
→ (lr-minimum-heapp (*data-seg2*) = lr-minimum-heapp (*data-seg1*))

EVENT: Disable lr-minimum-heapp-same-signature.

THEOREM: put-not-listp
((¬ listp (*list1*)) ∧ (¬ listp (*list2*)))
→ (put (*val*, *n*, *list1*) = put (*val*, *n*, *list2*))

THEOREM: put-zero
put (*val*, *n*, 0) = put (*val*, *n*, **nil**)

EVENT: Disable put-zero.

THEOREM: put-put
((*offset1* ∈ **N**) ∧ (*offset2* ∈ **N**))
→ (put (*val1*, *offset1*, put (*val2*, *offset2*, *list*))
= **if** *offset1* = *offset2* **then** put (*val1*, *offset1*, *list*)
else put (*val2*, *offset2*, put (*val1*, *offset1*, *list*)) **endif**)

THEOREM: proper-p-data-segmentp-implies-lr-proper-p-areasp
proper-p-data-segmentp (*data-seg*, *p*) → lr-proper-p-areasp (*data-seg*)

THEOREM: proper-p-statep-lr->p-implies-lr-proper-p-areasp
proper-p-statep (lr->p (*l*)) → lr-proper-p-areasp (p-data-segment (*l*))

EVENT: Disable proper-p-data-segmentp-implies-lr-proper-p-areasp.

THEOREM: lr-proper-free-listp-type-fetch-free-ptr
(lr-proper-free-listp (*data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*))
∧ lr-proper-p-areasp (*data-seg*))
→ (type (fetch (add-addr (fetch (identity (LR-FP-ADDR), *data-seg*),
identity (LR-REF-COUNT-OFFSET)),
data-seg))
≠ 'nat)

THEOREM: put-assoc-put-assoc-1
put-assoc (*val1*, *name*, put-assoc (*val2*, *name*, *alist*))
= put-assoc (*val1*, *name*, *alist*)

THEOREM: put-assoc-put-assoc-2
put-assoc (*val1*, *name1*, put-assoc (*val2*, *name2*, *alist*))
= **if** *name1* = *name2* **then** put-assoc (*val1*, *name1*, *alist*)
else put-assoc (*val2*, *name2*, put-assoc (*val1*, *name1*, *alist*)) **endif**

THEOREM: deposit-deposit
((offset (*addr1*) ∈ **N**) ∧ (offset (*addr2*) ∈ **N**))

```
→   (deposit (value1, addr1, deposit (value2, addr2, data-seg))
    =   if (area-name (addr1) = area-name (addr2))
          ∧   (offset (addr1) = offset (addr2))
        then deposit (value1, addr1, data-seg)
        else deposit (value2,
                      addr2,
                      deposit (value1, addr1, data-seg)) endif)
```

THEOREM: deposit-ref-count-move-outward

```
(offset (addr) ∈ N)
→   (deposit (value1,
              addr,
              deposit (value2,
                       add-addr (addr, identity (LR-REF-COUNT-OFFSET)),
                       data-seg))
    =   deposit (value2,
                 add-addr (addr, LR-REF-COUNT-OFFSET),
                 deposit (value1, addr, data-seg)))
```

DEFINITION:

```
ihint-2 (flag, s, l, table, c)
=   if s-err-flag (s) ≠ 'run then t
    elseif flag = 'list
    then if s-pos (s) ≃ nil then t
         elseif listp (s-expr-list (s))
         then ihint-2 (t, s, l, table, c)
              ∧   ihint-2 ('list,
                           s-set-expr (s-eval (t, s, c), s, nx (s-pos (s))),
                           lr-eval (t, s->lr1 (s, l, table), c),
                           table,
                           c)
         else t endif
    elseif c ≃ 0 then t
    elseif litatom (s-expr (s)) then t
    elseif s-expr (s) ≃ nil then t
    elseif car (s-expr (s)) = 'if
    then let lrtest be lr-if-ok (lr-eval (t,
                                          s->lr1 (s-set-pos (s,
                                                             dv (s-pos (s),
                                                                 1)),
                                                  l,
                                                  table),
                                          c)),
             stest be s-eval (t, s-set-pos (s, dv (s-pos (s), 1)), c)
```

```
        in
        if p-psw (lrtest) = 'run
        then if top (p-temp-stk (lrtest)) ≠ LR-F-ADDR
                then ihint-2 (t,
                              s-set-pos (s, dv (s-pos (s), 1)),
                              l,
                              table,
                              c)
                     ∧   ihint-2 (t,
                                  s-set-expr (stest,
                                              s,
                                              dv (s-pos (s), 2)),
                                  lr-pop-tstk (lrtest),
                                  table,
                                  c)
                else ihint-2 (t,
                              s-set-pos (s, dv (s-pos (s), 1)),
                              l,
                              table,
                              c)
                     ∧   ihint-2 (t,
                                  s-set-expr (stest,
                                              s,
                                              dv (s-pos (s), 3)),
                                  lr-pop-tstk (lrtest),
                                  table,
                                  c) endif
        else ihint-2 (t,
                      s-set-pos (s, dv (s-pos (s), 1)),
                      l,
                      table,
                      c) endif endlet
elseif car (s-expr (s)) = S-TEMP-EVAL
then ihint-2 (t, s-set-pos (s, dv (s-pos (s), 1)), l, table, c)
elseif car (s-expr (s)) = S-TEMP-TEST
then if p-max-temp-stk-size (l) ≮ (2 + length (p-temp-stk (l)))
        then if lr-eval-temp-setp (s->lr1 (s, l, table)) then t
                else ihint-2 (t, s-set-pos (s, dv (s-pos (s), 1)), l, table, c) endif
        else t endif
elseif car (s-expr (s)) = S-TEMP-FETCH then t
elseif car (s-expr (s)) = 'quote then t
elseif s-err-flag (s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c))
        ≠   'run
then ihint-2 ('list, s-set-pos (s, dv (s-pos (s), 1)), l, table, c)
```

```
    elseif subrp (car (s-expr (s)))
    then ihint-2 ('list, s-set-pos (s, dv (s-pos (s), 1)), l, table, c)
    elseif litatom (car (s-expr (s)))
    then let s-arg-s be s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c),
             lr-arg-s be lr-eval ('list,
                                  s->lr1 (s-set-pos (s, dv (s-pos (s), 1)),
                                          l,
                                          table),
                                  c)
         in
         ihint-2 ('list, s-set-pos (s, dv (s-pos (s), 1)), l, table, c)
         ∧  ihint-2 (t,
                     s-fun-call-state (s-arg-s, car (s-expr (s))),
                     lr-funcall (s->lr1 (s, l, table), lr-arg-s),
                     table,
                     c − 1) endlet
    else t endif
```

```
Definition:
induct-hint-4 (x, temp-stk)
=   if listp (x) then induct-hint-4 (cdr (x), cdr (temp-stk))
    else t endif
```

```
Theorem: lr-check-result1-append
 lr-check-result1 (append (x, y), temp-stk, data-seg)
 =   (lr-check-result1 (x, temp-stk, data-seg)
       ∧  lr-check-result1 (y, restn (length (x), temp-stk), data-seg))
```

```
Theorem: lr-proper-heapp-opener-4
 lr-proper-heapp (data-seg)
 →   (adpp (untag (lr-max-node (data-seg)), data-seg)
       ∧  lr-boundary-nodep (lr-max-node (data-seg)))
```

```
Theorem: length-strip-cars
 length (strip-cars (temp-vars)) = length (temp-vars)
```

```
Theorem: definedp-lr-compile-programs
 definedp (name, lr-compile-programs (progs, const-table))
 =   definedp (name, progs)
```

```
Theorem: lr-valp-deposit-fetch-free-pointer-offset-helper-1
 ((type (fetch (add-addr (addr, identity (lr-ref-count-offset)), data-seg))
   =   'nat)
  ∧  lr-good-pointerp (addr, data-seg)
  ∧  lr-nodep (free-addr, data-seg)
```

$\wedge$ (offset (*addr*) = offset (*free-addr*)))
$\rightarrow$ (type (fetch (add-addr (*free-addr*, identity (LR-REF-COUNT-OFFSET)),
data-seg))
= 'nat)

THEOREM: lr-boundary-nodep-equal-plus-fact-zero
((type (*addr1*) = 'addr)
$\wedge$ (cddr (*addr1*) = **nil**)
$\wedge$ listp (*addr1*)
$\wedge$ listp (untag (*addr1*))
$\wedge$ (offset (*addr1*) $\in$ **N**)
$\wedge$ lr-boundary-nodep (*addr1*)
$\wedge$ (type (*addr2*) = 'addr)
$\wedge$ (cddr (*addr2*) = **nil**)
$\wedge$ listp (*addr2*)
$\wedge$ listp (untag (*addr2*))
$\wedge$ (offset (*addr2*) $\in$ **N**)
$\wedge$ lr-boundary-nodep (*addr2*)
$\wedge$ (area-name (*addr2*) = area-name (*addr1*))
$\wedge$ ($m <$ LR-NODE-SIZE))
$\rightarrow$ ((offset (*addr1*) = (m + offset (*addr2*)))
= (($m \simeq$ 0) $\wedge$ (*addr1* = *addr2*)))

THEOREM: lr-boundary-nodep-equal-plus-fact
((type (*addr1*) = 'addr)
$\wedge$ (cddr (*addr1*) = **nil**)
$\wedge$ listp (*addr1*)
$\wedge$ listp (untag (*addr1*))
$\wedge$ (offset (*addr1*) $\in$ **N**)
$\wedge$ lr-boundary-nodep (*addr1*)
$\wedge$ (type (*addr2*) = 'addr)
$\wedge$ (cddr (*addr2*) = **nil**)
$\wedge$ listp (*addr2*)
$\wedge$ listp (untag (*addr2*))
$\wedge$ (offset (*addr2*) $\in$ **N**)
$\wedge$ lr-boundary-nodep (*addr2*)
$\wedge$ ($n <$ LR-NODE-SIZE)
$\wedge$ ($m <$ LR-NODE-SIZE)
$\wedge$ (area-name (*addr1*) = area-name (*addr2*)))
$\rightarrow$ (((*n* + offset (*addr1*)) = (*m* + offset (*addr2*)))
= ((fix (*n*) = fix (*m*)) $\wedge$ (*addr1* = *addr2*)))

THEOREM: lr-valp-deposit-fetch-free-pointer-offset
((type (fetch (add-addr (*free-addr*, LR-REF-COUNT-OFFSET), *data-seg*)) $\neq$ 'nat)
$\wedge$ lr-nodep (*free-addr*, *data-seg*)

$\wedge$ $(n <$ LR-NODE-SIZE$)$
$\wedge$ lr-valp $(value,\ addr,\ data\text{-}seg)$
$\rightarrow$ lr-valp $(value,\ addr,$ deposit $(anything,$ add-addr $(free\text{-}addr,\ n),\ data\text{-}seg))$

EVENT: Disable lr-valp-deposit-fetch-free-pointer-offset.

EVENT: Disable lr-valp-deposit-fetch-free-pointer-offset-helper-1.

THEOREM: lr-valp-deposit-fetch-free-pointer
((type (fetch (add-addr $(free\text{-}addr,$ LR-REF-COUNT-OFFSET$),\ data\text{-}seg)) $\neq$ 'nat)
$\wedge$ lr-nodep $(free\text{-}addr,\ data\text{-}seg)$
$\wedge$ lr-valp $(value,\ addr,\ data\text{-}seg)$
$\rightarrow$ lr-valp $(value,\ addr,$ deposit $(anything,\ free\text{-}addr,\ data\text{-}seg))$

EVENT: Disable lr-valp-deposit-fetch-free-pointer.

THEOREM: not-equal-x-add1-add1-x
$(x = (1 + (1 + x))) =$ **f**

THEOREM: not-equal-x-add1-x
$(x = (1 + x)) =$ **f**

THEOREM: p-run-subr-preserves-lr-valp
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l))))
$\wedge$ lr-programs-properp $(l,\ table)$
$\wedge$ lr-programs-properp $(new\text{-}l,\ table)$
$\wedge$ listp (lr-expr (l))
$\wedge$ proper-p-statep (lr->p $(new\text{-}l)$)
$\wedge$ lr-proper-free-listp (p-data-segment $(new\text{-}l)$)
$\wedge$ adpp (untag (lr-max-node (p-data-segment $(new\text{-}l)$)), p-data-segment $(new\text{-}l)$)
$\wedge$ lr-boundary-nodep (lr-max-node (p-data-segment $(new\text{-}l)$))
$\wedge$ (p-psw $(new\text{-}l)$ = 'run)
$\wedge$ (p-psw (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p $(new\text{-}l)$, lr-return-pc (l)))))
= 'run)
$\wedge$ lr-valp $(value,\ addr,$ p-data-segment $(new\text{-}l))$
$\wedge$ (p-prog-segment (l) = p-prog-segment $(new\text{-}l)$))
$\rightarrow$ lr-valp $(value,$
$addr,$
p-data-segment (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p $(new\text{-}l)$,
lr-return-pc (l))))))

THEOREM: numberp-offset-sub-addr
offset (sub-addr (*addr*, *n*)) ∈ **N**

THEOREM: lr-free-list-nodes-deposit-non-ref-count
(lr-nodep (*addr*, *data-seg*)
∧ lr-nodep (*max-addr*, *data-seg*)
∧ (*offset* ≠ LR-REF-COUNT-OFFSET)
∧ (*offset* ∈ **N**)
∧ (*offset* < LR-NODE-SIZE))
→ (lr-free-list-nodes (*max-addr*,
deposit (*anything*, add-addr (*addr*, *offset*), *data-seg*))
= lr-free-list-nodes (*max-addr*, *data-seg*))

THEOREM: lr-nodep-member-lr-node-listp-adpp-untag-listp
(lr-node-listp (*list*, *data-seg*) ∧ (*node* ∈ *list*)) → listp (untag (*node*))

EVENT: Disable lr-nodep-member-lr-node-listp-adpp-untag-listp.

THEOREM: lr-nodep-member-lr-node-listp-adpp-untag-numberp-offset
(lr-node-listp (*list*, *data-seg*) ∧ (*node* ∈ *list*)) → (offset (*node*) ∈ **N**)

THEOREM: lr-nodep-member-lr-node-listp-lr-boundaryp-offsetp
(lr-node-listp (*list*, *data-seg*) ∧ (*node* ∈ *list*))
→ lr-boundary-offsetp (offset (*node*))

THEOREM: lr-check-free-nodes-deposit-non-ref-count
(lr-nodep (*addr2*, *data-seg*)
∧ lr-nodep (*max-addr*, *data-seg*)
∧ (*offset* ≠ LR-REF-COUNT-OFFSET)
∧ (*offset* ∈ **N**)
∧ (*offset* < LR-NODE-SIZE)
∧ lr-node-listp (*node-list*, *data-seg*))
→ (lr-check-free-nodes (*addr1*,
node-list,
deposit (*anything*, add-addr (*addr2*, *offset*), *data-seg*),
max-addr)
= lr-check-free-nodes (*addr1*, *node-list*, *data-seg*, *max-addr*))

EVENT: Disable lr-nodep-member-lr-node-listp-adpp-untag-numberp-offset.

THEOREM: adpp-deposit-other-area
(adp-name (*adp*) ≠ area-name (*addr*))
→ (adpp (*adp*, deposit (*anything*, *addr*, *data-seg*)) = adpp (*adp*, *data-seg*))

EVENT: Disable adpp-deposit-other-area.

THEOREM: length-deposit
length (cdr (assoc (*name*, deposit (*anything*, *addr*, *data-seg*))))
= **if** definedp (area-name (*addr*), *data-seg*)
then if area-name (*addr*) = *name*
then if offset (*addr*) < length (cdr (assoc (*name*, *data-seg*)))
then length (cdr (assoc (*name*, *data-seg*)))
else 1 + offset (*addr*) **endif**
else length (cdr (assoc (*name*, *data-seg*))) **endif**
else length (cdr (assoc (*name*, *data-seg*))) **endif**

THEOREM: same-signature-deposit
(adpp (untag (*addr*), *segment2*) ∧ lr-proper-p-areasp (*segment2*))
→ (same-signature (*segment1*, deposit (*anything*, *addr*, *segment2*))
= same-signature (*segment1*, *segment2*))

THEOREM: lr-max-node-same-signature
same-signature (*data-seg1*, *data-seg2*)
→ (lr-max-node (*data-seg2*) = lr-max-node (*data-seg1*))

EVENT: Disable lr-max-node-same-signature.

THEOREM: lr-max-node-deposit
(adpp (untag (*addr*), *data-seg*) ∧ lr-proper-p-areasp (*data-seg*))
→ (lr-max-node (deposit (*anything*, *addr*, *data-seg*))
= lr-max-node (*data-seg*))

THEOREM: not-adpp-untag-node-not-definedp-lr-heap-name
(¬ definedp (area-name (*addr*), *data-seg*))
→ (¬ adpp (untag (*addr*), *data-seg*))

THEOREM: sub-addr-area-name-offset-same
(listp (untag (*addr1*))
∧ (offset (*addr1*) ∈ **N**)
∧ (cddr (*addr1*) = **nil**)
∧ (type (*addr1*) = type (*addr2*)))
→ ((*addr1* = sub-addr (*addr2*, *n*))
= ((offset (*addr1*) = (offset (*addr2*) − *n*))
∧ (area-name (*addr1*) = area-name (*addr2*))))

THEOREM: lr-free-list-nodes-member-greater-offset
(offset (*addr*) ≮ offset (*max-addr*))
→ (*addr* ∉ lr-free-list-nodes (*max-addr*, *data-seg*))

THEOREM: lr-free-list-nodes-deposit-lr-ref-count-offset
((type (*addr*) = `'addr`)

$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ adpp (untag (*addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ (area-name (*addr*) = `'heap`)
$\wedge$ (type (*max-addr*) = `'addr`)
$\wedge$ adpp (untag (*max-addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*max-addr*)
$\wedge$ (area-name (*max-addr*) = `'heap`)
$\wedge$ (type (*ref-count*) = `'nat`)
$\wedge$ (untag (*ref-count*) $\in$ **N**))
$\rightarrow$ (lr-free-list-nodes (*max-addr*,
deposit (*ref-count*,
add-addr (*addr*,
identity (LR-REF-COUNT-OFFSET)),
data-seg))
= delete (*addr*, lr-free-list-nodes (*max-addr*, *data-seg*)))

EVENT: Disable lr-free-list-nodes-member-greater-offset.

EVENT: Disable not-adpp-untag-node-not-definedp-lr-heap-name.

DEFINITION:
no-duplicatesp (*list*)
= **if** listp (*list*)
then if car (*list*) $\in$ cdr (*list*) **then f**
else no-duplicatesp (cdr (*list*)) **endif**
else t endif

THEOREM: not-member-occurences-0
($x \notin z$) $\rightarrow$ (occurrences (*x*, *z*) = `0`)

EVENT: Disable not-member-occurences-0.

THEOREM: no-duplicatesp-occurences-1
(no-duplicatesp (*list*) $\wedge$ ($e \in$ *list*)) $\rightarrow$ (occurrences (*e*, *list*) = `1`)

THEOREM: no-duplicatesp-lr-free-list-nodes
no-duplicatesp (lr-free-list-nodes (*addr*, *data-seg*))

THEOREM: member-area-name-offset-same
((*addr1* $\in$ *node-list*)
$\wedge$ (offset (*addr1*) $\in$ **N**)
$\wedge$ (cddr (*addr1*) = **nil**)

$\wedge$ listp (untag (*addr1*))
$\wedge$ listp (untag (*addr2*))
$\wedge$ (offset (*addr2*) $\in$ **N**)
$\wedge$ (cddr (*addr2*) = **nil**)
$\wedge$ (type (*addr1*) = type (*addr2*))
$\wedge$ (area-name (*addr1*) = area-name (*addr2*))
$\wedge$ (offset (*addr2*) = offset (*addr1*)))
$\rightarrow$ (*addr2* $\in$ *node-list*)

THEOREM: lr-check-free-nodes-delete-deposit
(lr-check-free-nodes (*addr2*, *node-list*, *data-seg*, *max-addr*)
$\wedge$ (*addr1* $\notin$ *node-list*)
$\wedge$ lr-nodep (*addr1*, *data-seg*)
$\wedge$ lr-node-listp (*node-list*, *data-seg*)
$\wedge$ (type (*ref-count*) = `'nat`)
$\wedge$ (untag (*ref-count*) $\in$ **N**))
$\rightarrow$ lr-check-free-nodes (*addr2*,
node-list,
deposit (*ref-count*,
add-addr (*addr1*,
identity (LR-REF-COUNT-OFFSET)),
data-seg),
max-addr)

EVENT: Disable lr-check-free-nodes-delete-deposit.

EVENT: Disable member-area-name-offset-same.

THEOREM: lr-check-free-nodes-deposit-free-ptr
(adpp (identity (untag (LR-FP-ADDR)), *data-seg*)
$\wedge$ lr-node-listp (*node-list*, *data-seg*))
$\rightarrow$ (lr-check-free-nodes (*addr*,
node-list,
deposit (*anything*, identity (LR-FP-ADDR), *data-seg*),
max-addr)
= lr-check-free-nodes (*addr*, *node-list*, *data-seg*, *max-addr*))

THEOREM: lr-free-list-nodes-deposit-free-ptr
(lr-nodep (*max-addr*, *data-seg*) $\wedge$ adpp (identity (untag (LR-FP-ADDR)), *data-seg*))
$\rightarrow$ (lr-free-list-nodes (*max-addr*,
deposit (*anything*, identity (LR-FP-ADDR), *data-seg*))
= lr-free-list-nodes (*max-addr*, *data-seg*))

THEOREM: deposit-ref-count-move-inward-2

(lr-nodep (*addr*, *data-seg*)
 ∧ (*offset* ≠ LR-REF-COUNT-OFFSET)
 ∧ (*offset* ≯ 0)
 ∧ (*offset* < LR-NODE-SIZE))
→ (deposit (*any1*,
 add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)),
 deposit (*any2*, add-addr (*addr*, *offset*), *data-seg*))
 = deposit (*any2*,
 add-addr (*addr*, *offset*),
 deposit (*any1*,
 add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)),
 data-seg)))

EVENT: Disable deposit-ref-count-move-inward-2.

THEOREM: lr-free-list-nodes-deposit-lr-nodep
(lr-nodep (*addr*, *data-seg*) ∧ lr-nodep (*max-addr*, *data-seg*))
→ (lr-free-list-nodes (*max-addr*, deposit (*anything*, *addr*, *data-seg*))
 = lr-free-list-nodes (*max-addr*, *data-seg*))

THEOREM: lr-check-free-nodes-deposit-lr-nodep
(lr-nodep (*addr2*, *data-seg*)
 ∧ lr-nodep (*max-addr*, *data-seg*)
 ∧ lr-node-listp (*node-list*, *data-seg*))
→ (lr-check-free-nodes (*addr1*,
 node-list,
 deposit (*anything*, *addr2*, *data-seg*),
 max-addr)
 = lr-check-free-nodes (*addr1*, *node-list*, *data-seg*, *max-addr*))

THEOREM: same-signature-cons
same-signature (*data-seg1*, cons (*x*, *data-seg2*))
= **if** listp (*data-seg1*)
 then (signature (car (*data-seg1*)) = signature (*x*))
 ∧ same-signature (cdr (*data-seg1*), *data-seg2*)
 else f endif

THEOREM: same-signature-nil
(*data-seg1* ≃ **nil**)
→ (same-signature (*data-seg1*, *data-seg2*) = (*data-seg2* ≃ **nil**))

THEOREM: listp-put-assoc
listp (put-assoc (*val*, *name*, *alist*)) = listp (*alist*)

THEOREM: not-same-signature-deposit-too-large-addr
(definedp (area-name (*addr*), *data-seg2*)
 ∧ lr-proper-p-areasp (*data-seg2*)
 ∧ (offset (*addr*) ≮ length (value (area-name (*addr*), *data-seg1*))))
→ (¬ same-signature (*data-seg1*, deposit (*any*, *addr*, *data-seg2*)))

EVENT: Disable same-signature-cons.

EVENT: Disable same-signature-nil.

THEOREM: adpp-deposit-a-list
adpp (*adp*, *data-seg*) → adpp (*adp*, deposit-a-list (*list*, *addr2*, *data-seg*))

THEOREM: lr-proper-p-areasp-deposit-a-list
lr-proper-p-areasp (*data-seg*)
→ lr-proper-p-areasp (deposit-a-list (*list*, *addr*, *data-seg*))

THEOREM: definedp-deposit-a-list
definedp (*tag*, deposit-a-list (*list*, *addr*, *data-seg*)) = definedp (*tag*, *data-seg*)

THEOREM: sub1-plus-not-zerop-fact-1
(*x* ≠ 0) → ((((*y* + *x*) − 1) < *y*) = **f**)

THEOREM: not-adpp-untag-add-addr-adpp-untag
adpp (untag (*addr*), *data-seg*)
→ (adpp (untag (add-addr (*addr*, *n*)), *data-seg*)
 = ((offset (*addr*) + *n*)
 < length (cdr (assoc (area-name (*addr*), *data-seg*)))))

THEOREM: not-same-signature-deposit-a-list-too-large-addr
(definedp (area-name (*addr*), *data-seg2*)
 ∧ lr-proper-p-areasp (*data-seg2*)
 ∧ (offset (*addr*) ≮ length (value (area-name (*addr*), *data-seg1*))))
→ (same-signature (*data-seg1*, deposit-a-list (*list*, *addr*, *data-seg2*))
 = **if** listp (*list*) **then f**
 else same-signature (*data-seg1*, *data-seg2*) **endif**)

THEOREM: same-signature-deposit-a-list
(adpp (untag (*addr*), *data-seg2*)
 ∧ lr-proper-p-areasp (*data-seg2*)
 ∧ same-signature (*data-seg1*, *data-seg2*))
→ (same-signature (*data-seg1*, deposit-a-list (*list*, *addr*, *data-seg2*))
 = ((offset (*addr*) + (length (*list*) − 1))
 < length (cdr (assoc (area-name (*addr*), *data-seg1*)))))

EVENT: Disable not-same-signature-deposit-too-large-addr.

EVENT: Disable not-same-signature-deposit-a-list-too-large-addr.

THEOREM: deposit-good-node-preserves-lr-proper-free-listp
(lr-proper-free-listp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*))
 ∧ (offset (fetch (LR-FP-ADDR, *data-seg*))
    < (((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) − 1) − 1) − 1))
 ∧ (type (*tag*) = 'nat)
 ∧ (type (*ref-count*) = 'nat)
 ∧ (untag (*ref-count*) ∈ **N**))
→ lr-proper-free-listp (deposit (fetch (add-addr (fetch (identity (LR-FP-ADDR),
data-seg),
identity (LR-REF-COUNT-OFFSET)),
data-seg),
identity (LR-FP-ADDR),
deposit-a-list (list (*tag*, *ref-count*, *x*, *y*),
fetch (identity (LR-FP-ADDR),
data-seg),
data-seg)))

THEOREM: p-run-subr-preserves-lr-proper-free-listp
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ lr-programs-properp (*new-l*, *table*)
 ∧ listp (lr-expr (*l*))
 ∧ proper-p-statep (lr->p (*new-l*))
 ∧ lr-proper-free-listp (p-data-segment (*new-l*))
 ∧ adpp (untag (lr-max-node (p-data-segment (*new-l*))), p-data-segment (*new-l*))
 ∧ lr-boundary-nodep (lr-max-node (p-data-segment (*new-l*)))
 ∧ (p-psw (*new-l*) = 'run)
 ∧ (p-psw (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*), lr-return-pc (*l*))))
    = 'run)
 ∧ (p-prog-segment (*l*) = p-prog-segment (*new-l*))
 ∧ (area-name (p-pc (*l*)) = area-name (p-pc (*new-l*))))
→ lr-proper-free-listp (p-data-segment (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*),
lr-return-pc (*l*))))))

EVENT: Disable same-signature-deposit.

THEOREM: lr-apply-subr-preserves-lr-proper-free-listp
let *new-l* **be** lr-eval ('`list`, lr-set-pos (l, *pos*), c)
in
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧ lr-programs-properp (l, *table*)
 ∧ listp (lr-expr (l))
 ∧ subrp (car (lr-expr (l)))
 ∧ (car (lr-expr (l)) ≠ '`if`)
 ∧ proper-p-statep (lr->p (*new-l*))
 ∧ lr-proper-free-listp (p-data-segment (*new-l*))
 ∧ adpp (untag (lr-max-node (p-data-segment (*new-l*))),
 p-data-segment (*new-l*))
 ∧ lr-boundary-nodep (lr-max-node (p-data-segment (*new-l*)))
 ∧ (p-psw (*new-l*) = '`run`)
 ∧ (p-psw (lr-apply-subr (l, *new-l*)) = '`run`))
→ lr-proper-free-listp (p-data-segment (lr-apply-subr (l, *new-l*))) **endlet**

THEOREM: lr-eval-preserves-proper-p-statep-lr->p-rewrite
(proper-p-statep (lr->p (l))
 ∧ good-posp (*flag*, offset (p-pc (l)), program-body (p-current-program (l)))
 ∧ lr-programs-properp (l, *table*)
 ∧ (p-psw (lr-eval (*flag*, l, c)) = '`run`))
→ proper-p-statep (lr->p (lr-eval (*flag*, l, c)))

THEOREM: lr-eval-preserves-cdr-p-ctrl-stk
(proper-p-statep (lr->p (l))
 ∧ good-posp (*flag*, offset (p-pc (l)), program-body (p-current-program (l)))
 ∧ lr-programs-properp (l, *table*)
 ∧ (p-psw (lr-eval (*flag*, l, c)) = '`run`))
→ (cdr (p-ctrl-stk (lr-eval (*flag*, l, c))) = cdr (p-ctrl-stk (l)))

THEOREM: lr-eval-preserves-strip-cars-bindings-car-p-ctrl-stk
(proper-p-statep (lr->p (l))
 ∧ good-posp (*flag*, offset (p-pc (l)), program-body (p-current-program (l)))
 ∧ lr-programs-properp (l, *table*)
 ∧ (p-psw (lr-eval (*flag*, l, c)) = '`run`))
→ (strip-cars (bindings (car (p-ctrl-stk (lr-eval (*flag*, l, c))))))
 = strip-cars (bindings (car (p-ctrl-stk (l)))))

THEOREM: lr-eval-preserves-lr-max-node
(proper-p-statep (lr->p (l))
 ∧ good-posp (*flag*, offset (p-pc (l)), program-body (p-current-program (l)))
 ∧ lr-programs-properp (l, *table*)
 ∧ (p-psw (lr-eval (*flag*, l, c)) = '`run`))
→ (lr-max-node (p-data-segment (lr-eval (*flag*, l, c)))
 = lr-max-node (p-data-segment (l)))

THEOREM: lr-eval-preserves-adpp
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = 'run))
→ (adpp (*adp*, p-data-segment (lr-eval (*flag*, *l*, *c*)))
 = adpp (*adp*, p-data-segment (*l*)))

THEOREM: lr-eval-preserves-length-assoc-data-segment
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = 'run))
→ (length (cdr (assoc (*name*, p-data-segment (lr-eval (*flag*, *l*, *c*))))))
 = length (cdr (assoc (*name*, p-data-segment (*l*)))))

THEOREM: lr-eval-preserves-proper-p-statep-lr->p-lr-set-pos
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, *pos*, program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)) = 'run))
→ proper-p-statep (lr->p (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)))

THEOREM: lr-eval-preserves-strip-cars-bindings-car-p-ctrl-stk-lr-set-pos
(proper-p-statep (lr->p (*l*))
 ∧ good-posp1 (*pos*, program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)) = 'run))
→ (strip-cars (bindings (car (p-ctrl-stk (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*))))))
 = strip-cars (bindings (car (p-ctrl-stk (*l*)))))

THEOREM: lr-eval-preserves-cdr-p-ctrl-stk-lr-set-pos
(proper-p-statep (lr->p (*l*))
 ∧ good-posp1 (*pos*, program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)) = 'run))
→ (cdr (p-ctrl-stk (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)))
 = cdr (p-ctrl-stk (*l*)))

THEOREM: lr-eval-preserves-adpp-lr-set-pos
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, *pos*, program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)) = 'run))
→ (adpp (*adp*, p-data-segment (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)))
 = adpp (*adp*, p-data-segment (*l*)))

THEOREM: lr-eval-preserves-lr-max-node-lr-set-pos
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, *pos*, program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)) = 'run))
→ (lr-max-node (p-data-segment (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)))
 = lr-max-node (p-data-segment (*l*)))

THEOREM: lr-eval-preserves-lr-proper-free-listp
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = 'run)
 ∧ adpp (untag (lr-max-node (p-data-segment (*l*))), p-data-segment (*l*))
 ∧ lr-boundary-nodep (lr-max-node (p-data-segment (*l*)))
 ∧ lr-proper-free-listp (p-data-segment (*l*)))
→ lr-proper-free-listp (p-data-segment (lr-eval (*flag*, *l*, *c*)))

THEOREM: lr-apply-subr-preserves-lr-valp
let *new-l* **be** lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)
in
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ listp (lr-expr (*l*))
 ∧ subrp (car (lr-expr (*l*)))
 ∧ (car (lr-expr (*l*)) ≠ 'if)
 ∧ proper-p-statep (lr->p (*new-l*))
 ∧ lr-proper-free-listp (p-data-segment (*new-l*))
 ∧ adpp (untag (lr-max-node (p-data-segment (*new-l*))),
 p-data-segment (*new-l*))
 ∧ lr-boundary-nodep (lr-max-node (p-data-segment (*new-l*)))
 ∧ (p-psw (*new-l*) = 'run)
 ∧ (p-psw (lr-apply-subr (*l*, *new-l*)) = 'run)
 ∧ lr-valp (*value*, *addr*, p-data-segment (*new-l*)))
→ lr-valp (*value*, *addr*, p-data-segment (lr-apply-subr (*l*, *new-l*))) **endlet**

THEOREM: lr-eval-preserves-lr-proper-free-listp-lr-set-pos
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, *pos*, program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)) = 'run)
 ∧ adpp (untag (lr-max-node (p-data-segment (*l*))), p-data-segment (*l*))
 ∧ lr-boundary-nodep (lr-max-node (p-data-segment (*l*)))
 ∧ lr-proper-free-listp (p-data-segment (*l*)))
→ lr-proper-free-listp (p-data-segment (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)))

THEOREM: lr-eval-preserves-lr-valp
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = 'run)
 ∧ lr-proper-free-listp (p-data-segment (*l*))
 ∧ adpp (untag (lr-max-node (p-data-segment (*l*))), p-data-segment (*l*))
 ∧ lr-boundary-nodep (lr-max-node (p-data-segment (*l*)))
 ∧ lr-valp (*value*, *addr*, p-data-segment (*l*)))
→ lr-valp (*value*, *addr*, p-data-segment (lr-eval (*flag*, *l*, *c*)))

THEOREM: lr-check-f-addrp-deposit-anything-anywhere
lr-check-f-addrp (*addr*, deposit (*anything*, *anywhere*, *data-seg*))
= lr-check-f-addrp (*addr*, *data-seg*)

THEOREM: lr-check-undef-addrp-deposit-anything-anywhere
lr-check-undef-addrp (*addr*, deposit (*anything*, *anywhere*, *data-seg*))
= lr-check-undef-addrp (*addr*, *data-seg*)

THEOREM: lr-check-listp-addrp-deposit-free-ptr-0
lr-check-listp-addrp (*addr*, deposit (*any*, identity (LR-FP-ADDR), *data-seg*))
= lr-check-listp-addrp (*addr*, *data-seg*)

THEOREM: lr-check-numberp-addrp-deposit-free-ptr-0
lr-check-numberp-addrp (*addr*, deposit (*any*, identity (LR-FP-ADDR), *data-seg*))
= lr-check-numberp-addrp (*addr*, *data-seg*)

THEOREM: lr-proper-heapp-nodep-deposit-free-ptr-0
lr-proper-heapp-nodep (*addr*, deposit (*any*, identity (LR-FP-ADDR), *data-seg*))
= lr-proper-heapp-nodep (*addr*, *data-seg*)

THEOREM: lr-proper-heapp2-deposit-free-ptr-0
lr-proper-heapp2 (*addr*, deposit (*any*, identity (LR-FP-ADDR), *data-seg*))
= lr-proper-heapp2 (*addr*, *data-seg*)

THEOREM: lr-boundary-offsetp-equal-plus-fact-zero
(lr-boundary-offsetp (*offset1*)
 ∧ lr-boundary-offsetp (*offset2*)
 ∧ (*n* < LR-NODE-SIZE)
 ∧ (*offset1* ∈ **N**)
 ∧ (*offset2* ∈ **N**))
→ (((*n* + *offset1*) = *offset2*)
 = ((*n* ≃ 0) ∧ (*offset1* = *offset2*)))

THEOREM: fetch-add-addr-deposit-a-list-node

(adpp (untag (*max-addr*), *data-seg*)
 ∧ lr-boundary-nodep (*max-addr*)
 ∧ adpp (untag (*addr*), *data-seg*)
 ∧ lr-boundary-nodep (*addr*)
 ∧ (area-name (*addr*) = area-name (*max-addr*))
 ∧ (*n* < LR-NODE-SIZE))
→ (fetch (add-addr (*max-addr*, *n*),
 deposit-a-list (list (*x0*, *x1*, *x2*, *x3*), *addr*, *data-seg*))
 = **if** offset (*addr*) = offset (*max-addr*)
 then get (*n*, list (*x0*, *x1*, *x2*, *x3*))
 else fetch (add-addr (*max-addr*, *n*), *data-seg*) **endif**)

THEOREM: fetch-deposit-a-list-node
(adpp (untag (*max-addr*), *data-seg*)
 ∧ lr-boundary-nodep (*max-addr*)
 ∧ adpp (untag (*addr*), *data-seg*)
 ∧ lr-boundary-nodep (*addr*)
 ∧ (area-name (*addr*) = area-name (*max-addr*)))
→ (fetch (*max-addr*, deposit-a-list (list (*x0*, *x1*, *x2*, *x3*), *addr*, *data-seg*))
 = **if** offset (*addr*) = offset (*max-addr*) **then** *x0*
 else fetch (*max-addr*, *data-seg*) **endif**)

EVENT: Disable lr-boundary-offsetp-equal-plus-fact-zero.

THEOREM: lr-check-f-addrp-deposit-a-list
lr-check-f-addrp (*addr*, deposit-a-list (*list*, *anywhere*, *data-seg*))
= lr-check-f-addrp (*addr*, *data-seg*)

THEOREM: lr-check-undef-addrp-deposit-a-list
lr-check-undef-addrp (*addr*, deposit-a-list (*list*, *anywhere*, *data-seg*))
= lr-check-undef-addrp (*addr*, *data-seg*)

THEOREM: lr-check-numberp-addrp-deposit-a-list-cons
((offset (*addr*) ≠ offset (*max-addr*))
 ∧ (type (*max-addr*) = `'addr`)
 ∧ (cddr (*max-addr*) = **nil**)
 ∧ listp (*max-addr*)
 ∧ adpp (untag (*max-addr*), *data-seg*)
 ∧ lr-boundary-nodep (*max-addr*)
 ∧ (area-name (*max-addr*) = LR-HEAP-NAME)
 ∧ (type (*addr*) = `'addr`)
 ∧ (cddr (*addr*) = **nil**)
 ∧ listp (*addr*)
 ∧ adpp (untag (*addr*), *data-seg*)

$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ (area-name (*addr*) = LR-HEAP-NAME)
$\wedge$ lr-check-numberp-addrp (*max-addr*, *data-seg*)
$\wedge$ (type (*tag*) = `'nat`)
$\wedge$ (untag (*tag*) $\in$ **N**))
$\rightarrow$ lr-check-numberp-addrp (*max-addr*,
deposit-a-list (list (*x0*, *tag*, *x2*, *x3*),
addr,
data-seg))

THEOREM: lr-check-listp-addrp-deposit-a-list-cons
(lr-good-pointerp (*good-pointer1*, *data-seg*)
$\wedge$ lr-good-pointerp (*good-pointer2*, *data-seg*)
$\wedge$ adpp (untag (*addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ adpp (untag (*max-addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*max-addr*)
$\wedge$ (area-name (*addr*) = LR-HEAP-NAME)
$\wedge$ (area-name (*max-addr*) = LR-HEAP-NAME)
$\wedge$ (type (*tag*) = `'nat`)
$\wedge$ (untag (*tag*) $\in$ **N**)
$\wedge$ (offset (*addr*) = offset (*max-addr*)))
$\rightarrow$ lr-check-listp-addrp (*max-addr*,
deposit-a-list (list (identity (tag (`'nat`,
LR-CONS-TAG)),
tag,
good-pointer1,
good-pointer2),
addr,
data-seg))

THEOREM: lr-check-listp-addrp-deposit-a-list-other-place
(adpp (untag (*addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ adpp (untag (*max-addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*max-addr*)
$\wedge$ (area-name (*addr*) = LR-HEAP-NAME)
$\wedge$ (area-name (*max-addr*) = LR-HEAP-NAME)
$\wedge$ lr-check-listp-addrp (*max-addr*, *data-seg*)
$\wedge$ (offset (*addr*) $\neq$ offset (*max-addr*))
$\wedge$ (type (*ref-count*) = `'nat`))
$\rightarrow$ lr-check-listp-addrp (*max-addr*,
deposit-a-list (list (*x0*, *ref-count*, *x2*, *x3*),
addr,
data-seg))

THEOREM: lr-proper-heapp-nodep-deposit-a-list-cons
(lr-nodep (*max-addr*, *data-seg*)
 ∧ lr-nodep (*addr*, *data-seg*)
 ∧ lr-proper-heapp-nodep (*max-addr*, *data-seg*)
 ∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) ≠ 'nat)
 ∧ lr-good-pointerp (*good-pointer1*, *data-seg*)
 ∧ lr-good-pointerp (*good-pointer2*, *data-seg*)
 ∧ (type (*tag*) = 'nat)
 ∧ (untag (*tag*) ∈ **N**))
→ lr-proper-heapp-nodep (*max-addr*,
deposit-a-list (list (identity (tag ('nat,
LR-CONS-TAG)),
tag,
good-pointer1,
good-pointer2),
addr,
data-seg))

THEOREM: lr-proper-heapp2-deposit-a-list-cons
(lr-nodep (*max-addr*, *data-seg*)
 ∧ lr-nodep (*addr*, *data-seg*)
 ∧ lr-proper-heapp2 (*max-addr*, *data-seg*)
 ∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) ≠ 'nat)
 ∧ lr-good-pointerp (*good-pointer1*, *data-seg*)
 ∧ lr-good-pointerp (*good-pointer2*, *data-seg*)
 ∧ (type (*tag*) = 'nat)
 ∧ (untag (*tag*) ∈ **N**))
→ lr-proper-heapp2 (*max-addr*,
deposit-a-list (list (identity (tag ('nat, LR-CONS-TAG)),
tag,
good-pointer1,
good-pointer2),
addr,
data-seg))

THEOREM: not-psw-run-lr-eval
(p-psw (*l*) ≠ 'run) → (lr-eval (*flag*, *l*, *c*) = *l*)

THEOREM: program-body-assoc-lr-compile-programs
program-body (assoc (*name*, lr-compile-programs (*progs*, *table*)))
= lr-compile-body (**t**,
s-body (assoc (*name*, *progs*)),
lr-make-temp-name-alist (s-temp-list (assoc (*name*, *progs*)),
s-formals (assoc (*name*, *progs*))),
table)

THEOREM: listp-lr-compile-body
listp (lr-compile-body (*flag*, *body*, *temp-name-alist*, *table*)) = listp (*body*)

THEOREM: car-lr-compile-body
(*flag* ≠ 'list)
→ (car (lr-compile-body (*flag*, *body*, *temp-name-alist*, *table*)) = car (*body*))

THEOREM: good-posp1-expand-list-temps
(((*temp* = S-TEMP-EVAL) ∨ (*temp* = S-TEMP-TEST)) ∧ listp (*pos*))
→ (good-posp1 (*pos*, list (*temp*, *body*, *name*))
= ((car (*pos*) = 1) ∧ good-posp1 (cdr (*pos*), *body*)))

THEOREM: length-lr-compile-body-list
length (lr-compile-body ('list, *body*, *temp-name-alist*, *table*))
= length (*body*)

THEOREM: get-lr-compile-body-list
get (*n*, lr-compile-body ('list, *body*, *temp-name-alist*, *table*))
= lr-compile-body (**t**, get (*n*, *body*), *temp-name-alist*, *table*)

THEOREM: get-lr-compile-body
(listp (*body*)
∧ (car (*body*) ≠ S-TEMP-FETCH)
∧ (car (*body*) ≠ S-TEMP-EVAL)
∧ (car (*body*) ≠ S-TEMP-TEST)
∧ (car (*body*) ≠ 'quote)
∧ (*n* ≯ 0))
→ (get (*n*, lr-compile-body (**t**, *body*, *temp-name-alist*, *table*))
= lr-compile-body (**t**, get (*n*, *body*), *temp-name-alist*, *table*))

THEOREM: length-lr-compile-body-t
(listp (*body*)
∧ (car (*body*) ≠ S-TEMP-FETCH)
∧ (car (*body*) ≠ S-TEMP-EVAL)
∧ (car (*body*) ≠ S-TEMP-TEST)
∧ (car (*body*) ≠ 'quote))
→ (length (lr-compile-body (**t**, *body*, *temp-name-alist*, *table*))
= length (*body*))

THEOREM: good-posp1-lr-compile-body
good-posp1 (*pos*, lr-compile-body (**t**, *body*, *temp-name-alist*, *table*))
= good-posp1 (*pos*, *body*)

THEOREM: cur-expr-lr-compile-body-t
good-posp1 (*pos*, *body*)
→ (cur-expr (*pos*, lr-compile-body (**t**, *body*, *temp-name-alist*, *table*))
= lr-compile-body (**t**, cur-expr (*pos*, *body*), *temp-name-alist*, *table*))

THEOREM: lr-check-result1-singleton-list-opener
lr-check-result1 (list (*x*), *temp-stk*, *data-seg*)
= lr-valp (*x*, car (*temp-stk*), *data-seg*)

THEOREM: proper-p-temp-stkp-plistp-p-temp-stk
proper-p-temp-stkp (*temp-stk*, *p*) → plistp (*temp-stk*)

THEOREM: proper-p-statep-lr->p-plistp-p-temp-stk
proper-p-statep (lr->p (*l*)) → plistp (p-temp-stk (*l*))

THEOREM: proper-p-statep-lr->p-not-0-p-temp-stk
proper-p-statep (lr->p (*l*)) → (p-temp-stk (*l*) ≠ `0`)

THEOREM: plistp-lastcdr-nil
plistp (*list*) → (lastcdr (*list*) = **nil**)

THEOREM: lr-eval-preserves-lr-valp-lr-set-expr
(proper-p-statep (lr->p (*l*))
∧ proper-p-statep (lr->p (lr-set-expr (*l1*, *l*, *pos*)))
∧ good-posp (*flag*, *pos*, program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (lr-eval (*flag*, lr-set-expr (*l1*, *l*, *pos*), *c*)) = `'run`)
∧ lr-proper-free-listp (p-data-segment (*l*))
∧ lr-proper-free-listp (p-data-segment (*l1*))
∧ adpp (untag (lr-max-node (p-data-segment (*l*))), p-data-segment (*l*))
∧ lr-boundary-nodep (lr-max-node (p-data-segment (*l*)))
∧ adpp (untag (lr-max-node (p-data-segment (*l1*))), p-data-segment (*l1*))
∧ lr-boundary-nodep (lr-max-node (p-data-segment (*l1*)))
∧ lr-valp (*value*, *addr*, p-data-segment (*l1*))
∧ (length (cdr (assoc (LR-HEAP-NAME, p-data-segment (*l1*))))
= length (cdr (assoc (LR-HEAP-NAME, p-data-segment (*l*))))))
→ lr-valp (*value*,
addr,
p-data-segment (lr-eval (*flag*, lr-set-expr (*l1*, *l*, *pos*), *c*)))

THEOREM: lr-eval-preserves-proper-p-statep-lr->p-lr-set-expr
(proper-p-statep (lr->p (*l*))
∧ good-posp (*flag*, *pos*, program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (lr-eval (*flag*, lr-set-expr (*l1*, *l*, *pos*), *c*)) = `'run`)
∧ lr-programs-properp (*l1*, *table*)
∧ proper-p-statep (lr->p (*l1*))
∧ (cdr (p-ctrl-stk (*l1*)) = cdr (p-ctrl-stk (*l*)))
∧ (strip-cars (bindings (car (p-ctrl-stk (*l1*))))
= strip-cars (bindings (car (p-ctrl-stk (*l*))))))

$\wedge$ (p-prog-segment (*l1*) = p-prog-segment (*l*))
$\wedge$ (p-word-size (*l1*) = p-word-size (*l*))
$\wedge$ (p-max-ctrl-stk-size (*l1*) = p-max-ctrl-stk-size (*l*))
$\wedge$ (p-max-temp-stk-size (*l1*) = p-max-temp-stk-size (*l*)))
$\rightarrow$ proper-p-statep (lr->p (lr-eval (*flag*, lr-set-expr (*l1*, *l*, *pos*), *c*)))

THEOREM: lr-check-result-flag-list-cons-value
let *l2* **be** lr-eval (`'list`, lr-set-expr (lr-eval (**t**, *l*, *c*), *l*, nx (*pos*)), *c*)
in
(good-posp (`'list`, *pos*, program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ proper-p-statep (lr->p (*l*))
$\wedge$ listp (lr-expr-list (*l*))
$\wedge$ listp (offset (p-pc (*l*)))
$\wedge$ lr-proper-heapp (p-data-segment (*l*))
$\wedge$ lr-check-result (**t**,
value1,
p-temp-stk (lr-eval (**t**, *l*, *c*)),
p-data-segment (lr-eval (**t**, *l*, *c*)),
p-temp-stk (*l*))
$\wedge$ lr-check-result (`'list`,
value2,
p-temp-stk (*l2*),
p-data-segment (*l2*),
p-temp-stk (lr-eval (**t**, *l*, *c*)))
$\wedge$ (p-psw (*l2*) = `'run`)
$\wedge$ (*pos* = offset (p-pc (*l*)))
$\wedge$ (*temp-stk* = p-temp-stk (*l*)))
$\rightarrow$ lr-check-result (`'list`,
cons (*value1*, *value2*),
p-temp-stk (*l2*),
p-data-segment (*l2*),
temp-stk) **endlet**

THEOREM: lr-check-result-nil
lr-proper-heapp (*data-seg*)
$\rightarrow$ lr-check-result (`'list`, **nil**, *temp-stk*, *data-seg*, *temp-stk*)

THEOREM: litatom-lr-compile-body
litatom (lr-compile-body (**t**, *body*, *temp-name-alist*, *table*)) = litatom (*body*)

THEOREM: lr-params-lr-push-tstk
lr-params (*frame*, lr-push-tstk (*l*, *anything*)) = lr-params (*frame*, *l*)

THEOREM: lr-temps-lr-push-tstk
lr-temps (*frame*, lr-push-tstk (*l*, *anything*)) = lr-temps (*frame*, *l*)

THEOREM: program-body-p-current-program-s->lr1
program-body (p-current-program (s->lr1 (*s*, *l*, *table*)))
= lr-compile-body (**t**,
s-body (s-prog (*s*)),
lr-make-temp-name-alist (s-temp-list (s-prog (*s*)),
s-formals (s-prog (*s*))),
table)

THEOREM: name-car-lr-compile-programs-progs
name (car (lr-compile-programs (s-progs (*s*), *table*))) = caar (s-progs (*s*))

THEOREM: car-car-lr-compile-programs-progs
caar (lr-compile-programs (s-progs (*s*), *table*)) = caar (s-progs (*s*))

THEOREM: s-good-statep-program-body-car-lr-compile-programs
s-good-statep (*s*, *c*)
→ (program-body (car (lr-compile-programs (s-progs (*s*), *table*)))
= lr-compile-body (**t**,
s-body (car (s-progs (*s*))),
lr-make-temp-name-alist (s-temp-list (car (s-progs (*s*))),
s-formals (car (s-progs (*s*)))),
table))

THEOREM: good-posp-lr-compile-body
good-posp (*flag*, *pos*, lr-compile-body (**t**, *body*, *temp-name-alist*, *table*))
= good-posp (*flag*, *pos*, *body*)

THEOREM: strip-cars-lr-compile-programs
strip-cars (lr-compile-programs (*progs*, *table*)) = strip-cars (*progs*)

THEOREM: listp-lr-expr-list-s->lr1
good-posp (`'list`, s-pos (*s*), s-body (s-prog (*s*)))
→ (listp (lr-expr-list (s->lr1 (*s*, *l*, *table*))) = listp (s-expr-list (*s*)))

THEOREM: formal-vars-lr-compile-programs
formal-vars (assoc (*name*, lr-compile-programs (*progs*, *table*)))
= s-formals (assoc (*name*, *progs*))

THEOREM: formal-vars-p-current-program-s->lr1
formal-vars (p-current-program (s->lr1 (*s*, *l*, *table*))) = s-formals (s-prog (*s*))

THEOREM: temp-var-dcls-lr-compile-programs
definedp (*name*, *progs*)
→ (temp-var-dcls (assoc (*name*, lr-compile-programs (*progs*, *table*)))
= lr-make-temp-var-dcls (lr-make-temp-name-alist (s-temp-list (assoc (*name*,
progs)),
s-formals (assoc (*name*,
progs)))))

THEOREM: temp-var-dcls-assoc-p-current-program-s->lr1
definedp (s-pname (*s*), s-progs (*s*))
→ (temp-var-dcls (p-current-program (s->lr1 (*s*, *l*, *table*)))
= lr-make-temp-var-dcls (lr-make-temp-name-alist (s-temp-list (s-prog (*s*)),
s-formals (s-prog (*s*)))))

THEOREM: lr-set-expr-s->lr1-s-set-expr
s->lr1 (s-set-expr (s-eval (**t**, *s*, *c*), *s*, nx (s-pos (*s*))),
lr-eval (**t**, s->lr1 (*s*, *l*, *table*), *c*),
table)
= lr-set-error (lr-set-expr (lr-eval (**t**, s->lr1 (*s*, *l*, *table*), *c*),
s->lr1 (*s*, *l*, *table*),
nx (s-pos (*s*))),
s-err-flag (s-eval (**t**, *s*, *c*)))

THEOREM: p-current-program-lr-set-error
p-current-program (lr-set-error (*l*, *err-flag*)) = p-current-program (*l*)

THEOREM: lr-set-error-lr-set-error
lr-set-error (lr-set-error (*l*, *err-flag1*), *err-flag2*)
= lr-set-error (*l*, *err-flag2*)

THEOREM: proper-p-statep-lr-set-error
proper-p-statep (lr-set-error (*l*, *err-flag*)) = proper-p-statep (*l*)

THEOREM: lr-params-lr-set-error
lr-params (*frame*, lr-set-error (*l*, *err-flag*)) = lr-params (*frame*, *l*)

THEOREM: lr-temps-lr-set-error
lr-temps (*frame*, lr-set-error (*l*, *err-flag*)) = lr-temps (*frame*, *l*)

THEOREM: lr-s-similar-statesp-lr-set-error
lr-s-similar-statesp (*params*, *temps*, lr-set-error (*l*, *err-flag*), *table*)
= lr-s-similar-statesp (*params*, *temps*, *l*, *table*)

THEOREM: lr-s-similar-statesp-lr-set-expr
((area-name (p-pc (*l1*)) = area-name (p-pc (*l2*)))
∧ (p-prog-segment (*l1*) = p-prog-segment (*l2*)))
→ (lr-s-similar-statesp (*params*, *temps*, lr-set-expr (*l1*, *l2*, *pos*), *table*)
= lr-s-similar-statesp (*params*, *temps*, *l1*, *table*))

THEOREM: lr-eval-zerop-clock
((*c* ≃ 0) ∧ (*flag* ≠ 'list) ∧ (p-psw (*l*) = 'run))
→ (lr-eval (*flag*, *l*, *c*) = lr-set-error (*l*, 'out-of-time))

THEOREM: litatom-lr-expr-s->lr1
(good-posp1 (s-pos (s), s-body (s-prog (s))) ∧ litatom (s-expr (s)))
→ (lr-expr (s->lr1 (s, l, *table*)) = s-expr (s))

THEOREM: lr-eval-litatom-opener
(good-posp1 (s-pos (s), s-body (s-prog (s)))
∧ (*flag* ≠ 'list)
∧ (c ≉ 0)
∧ litatom (s-expr (s))
∧ (s-err-flag (s) = 'run))
→ (lr-eval (*flag*, s->lr1 (s, l, *table*), c)
= lr-push-tstk (s->lr1 (s, l, *table*),
local-var-value (s-expr (s), p-ctrl-stk (l))))

THEOREM: lr-s-similar-statesp-lr-push-tstk-litatom
lr-s-similar-statesp (*s-params*, *s-temps*, lr-push-tstk (l, *value*), *table*)
= lr-s-similar-statesp (*s-params*, *s-temps*, l, *table*)

THEOREM: lr-s-similar-params-assoc-definedp
(lr-s-similar-params (*s-params*, *lr-params*, *data-seg*)
∧ definedp (*name*, *lr-params*))
→ lr-valp (cdr (assoc (*name*, *s-params*)), cdr (assoc (*name*, *lr-params*)), *data-seg*)

THEOREM: proper-p-statep-lr->p-strip-cars-bindings-ctrl-stk
(proper-p-statep (lr->p (l))
∧ definedp (area-name (p-pc (l)), p-prog-segment (l)))
→ (strip-cars (bindings (car (p-ctrl-stk (l))))
= append (formal-vars (assoc (area-name (p-pc (l)), p-prog-segment (l))),
strip-cars (temp-var-dcls (assoc (area-name (p-pc (l)),
p-prog-segment (l))))))

DEFINITION:
induct-hint-11 (v, y)
= **if** listp (v)
then if listp (y) **then** induct-hint-11 (cdr (v), cdr (y))
else t endif
else t endif

THEOREM: equal-append-same-length-fact
(length (v) = length (y))
→ ((append (strip-cars (v), w) = append (y, z))
= ((strip-cars (v) = plist (y)) ∧ (w = z)))

THEOREM: definedp-strip-cars-append-member-x
(strip-cars (x) = append (y, z))
→ ((e ∈ y) = definedp (e, firstn (length (y), x)))

THEOREM: proper-p-statep-lr->p-member-formals-definedp-bindings
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ definedp (s-pname (*s*), s-progs (*s*))
 ∧ (*x* ∈ s-formals (assoc (s-pname (*s*), s-progs (*s*)))))
→ definedp (*x*,
 firstn (length (s-formals (assoc (s-pname (*s*), s-progs (*s*)))),
 bindings (car (p-ctrl-stk (*l*)))))

THEOREM: lr-valp-addr-0
¬ lr-valp (*addr*, 0, *data-seg*)

THEOREM: lr-valp-cdr-assoc-firstn-cdr-assoc
lr-valp (*addr*, cdr (assoc (*name*, firstn (*n*, *list*))), *data-seg*)
→ lr-valp (*addr*, cdr (assoc (*name*, *list*)), *data-seg*)

THEOREM: lr-s-similar-statesp-lr-s-similar-params-opener
(lr-s-similar-statesp (*s-params*, *s-temps*, *l*, *table*)
 ∧ (*frame* = car (p-ctrl-stk (*l*)))
 ∧ (*data-seg* = p-data-segment (*l*)))
→ lr-s-similar-params (*s-params*, lr-params (*frame*, *l*), *data-seg*)

THEOREM: lr-s-similar-statesp-lr-s-similar-temps-opener
(lr-s-similar-statesp (*s-params*, *s-temps*, *l*, *table*)
 ∧ (*frame* = car (p-ctrl-stk (*l*)))
 ∧ (*data-seg* = p-data-segment (*l*)))
→ lr-s-similar-temps (*s-temps*, lr-temps (*frame*, *l*), *data-seg*)

THEOREM: strip-cars-lr-make-temp-var-dcls
strip-cars (lr-make-temp-var-dcls (*temp-alist*)) = strip-cdrs (*temp-alist*)

THEOREM: lr-check-result-lr-push-tstk
let *value* **be** cdr (assoc (s-expr (*s*), bindings (car (p-ctrl-stk (*l*)))))
in
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
 ∧ definedp (s-pname (*s*), s-progs (*s*))
 ∧ (s-err-flag (*s*) = 'run)
 ∧ litatom (s-expr (*s*))
 ∧ proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ (p-psw (lr-push-tstk (s->lr1 (*s*, *l*, *table*), *value*)) = 'run)
 ∧ lr-s-similar-statesp (s-params (*s*),
 s-temps (*s*),
 s->lr1 (*s*, *l*, *table*),
 table))

→ lr-check-result (**t**,
cdr (assoc (s-expr (*s*), s-params (*s*))),
p-temp-stk (lr-push-tstk (s->lr1 (*s*, *l*, *table*),
value)),
p-data-segment (*l*),
p-temp-stk (*l*)) **endlet**

THEOREM: s->lr1-s-set-pos-lr-set-pos
s->lr1 (s-set-pos (*s*, *pos*), *l*, *table*) = lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*)

THEOREM: lr-params-lr-set-pos
lr-params (*frame*, lr-set-pos (*l*, *pos*)) = lr-params (*frame*, *l*)

THEOREM: lr-temps-lr-set-pos
lr-temps (*frame*, lr-set-pos (*l*, *pos*)) = lr-temps (*frame*, *l*)

THEOREM: lr-s-similar-statesp-lr-s-set-pos
lr-s-similar-statesp (*s-params*, *s-temps*, lr-set-pos (*l*, *pos*), *table*)
= lr-s-similar-statesp (*s-params*, *s-temps*, *l*, *table*)

THEOREM: lr-set-expr-s->lr1-s-set-expr-lr-pop-tstk
s->lr1 (s-set-expr (s-eval (**t**, s-set-pos (*s*, *pos*), *c*), *s*, dv (s-pos (*s*), *n*)),
lr-pop-tstk (lr-if-ok (lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*))),
table)
= lr-set-error (lr-set-expr (lr-pop-tstk (lr-if-ok (lr-eval (**t**,
lr-set-pos (s->lr1 (*s*,
l,
table),
pos),
c))),
s->lr1 (*s*, *l*, *table*),
dv (s-pos (*s*), *n*)),
s-err-flag (s-eval (**t**, s-set-pos (*s*, *pos*), *c*)))

THEOREM: lr-s-similar-statesp-lr-pop-tstk
lr-s-similar-statesp (*s-params*, *s-temps*, lr-pop-tstk (*l*), *table*)
= lr-s-similar-statesp (*s-params*, *s-temps*, *l*, *table*)

THEOREM: listp-lr-expr-s->lr1
good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
→ (listp (lr-expr (s->lr1 (*s*, *l*, *table*))) = listp (s-expr (*s*)))

THEOREM: litatom-lr-expr-s->lr1-s-expr
good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
→ (litatom (lr-expr (s->lr1 (*s*, *l*, *table*))) = litatom (s-expr (*s*)))

THEOREM: car-lr-expr-s->lr1
good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
→ (car (lr-expr (s->lr1 (*s*, *l*, *table*))) = car (s-expr (*s*)))

THEOREM: equal-p-psw-lr-eval-run-lr-eval-lr-set-error
(p-psw (*l*) = 'run)
→ (lr-eval (*flag*, lr-set-error (*l*, 'run), *c*) = lr-eval (*flag*, *l*, *c*))

THEOREM: lr-proper-heapp-lr-good-pointerp-lr-proper-heapp-nodep
(lr-good-pointerp (*addr*, *data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ lr-proper-heapp (*data-seg*))
→ lr-proper-heapp-nodep (*addr*, *data-seg*)

THEOREM: lr-check-result-f-not-lr-f-addr
((car (*temp-stk*) ≠ LR-F-ADDR)
∧ lr-proper-p-areasp (*data-seg*)
∧ listp (*temp-stk*))
→ (lr-check-result (**t**, **f**, *temp-stk*, *data-seg*, *orig-temp-stk*) = **f**)

THEOREM: lr-check-result-t-chain
((*flag* ≠ 'list)
∧ lr-check-result (**t**, *ans*, *temp-stk2*, *data-seg2*, cdr (*temp-stk1*))
∧ lr-check-result (**t**, *anything*, *temp-stk1*, *data-seg1*, *temp-stk0*))
→ lr-check-result (*flag*, *ans*, *temp-stk2*, *data-seg2*, *temp-stk0*)

THEOREM: lr-check-result-not-f-lr-f-addr
((car (*temp-stk*) = LR-F-ADDR)
∧ listp (*temp-stk*)
∧ lr-proper-p-areasp (*data-seg*)
∧ (*ans* ≠ **f**))
→ (lr-check-result (**t**, *ans*, *temp-stk*, *data-seg*, *orig-temp-stk*) = **f**)

THEOREM: lr-eval-leaves-listp-p-temp-stk
(proper-p-statep (lr->p (*l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ (*flag* ≠ 'list)
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = 'run))
→ listp (p-temp-stk (lr-eval (*flag*, *l*, *c*)))

THEOREM: lr-eval-s->lr1-if-opener-1
let *lr-test* **be** lr-if-ok (lr-eval (**t**,
lr-set-pos (s->lr1 (*s*, *l*, *table*),
dv (s-pos (*s*), 1)),

c))
in
(good-posp1 (s-pos (s), s-body (s-prog (s)))
 ∧ listp (s-expr (s))
 ∧ (car (s-expr (s)) = `'if`)
 ∧ ($flag \neq$ `'list`)
 ∧ (p-psw ($lr\text{-}test$) = `'run`)
 ∧ (top (p-temp-stk ($lr\text{-}test$)) ≠ LR-F-ADDR))
→ (lr-eval ($flag$, s->lr1 (s, l, $table$), c)
 = lr-eval (**t**,
 lr-set-expr (lr-pop-tstk ($lr\text{-}test$),
 s->lr1 (s, l, $table$),
 dv (s-pos (s), `2`)),
 c)) **endlet**

THEOREM: lr-eval-s->lr1-if-opener-2
let $lr\text{-}test$ **be** lr-if-ok (lr-eval (**t**,
 lr-set-pos (s->lr1 (s, l, $table$),
 dv (s-pos (s), `1`)),
 c))
in
(good-posp1 (s-pos (s), s-body (s-prog (s)))
 ∧ listp (s-expr (s))
 ∧ (car (s-expr (s)) = `'if`)
 ∧ ($flag \neq$ `'list`)
 ∧ (p-psw ($lr\text{-}test$) = `'run`)
 ∧ (top (p-temp-stk ($lr\text{-}test$)) = LR-F-ADDR))
→ (lr-eval ($flag$, s->lr1 (s, l, $table$), c)
 = lr-eval (**t**,
 lr-set-expr (lr-pop-tstk ($lr\text{-}test$),
 s->lr1 (s, l, $table$),
 dv (s-pos (s), `3`)),
 c)) **endlet**

THEOREM: lr-eval-s->lr1-if-opener-3
(good-posp1 (s-pos (s), s-body (s-prog (s)))
 ∧ ($flag \neq$ `'list`)
 ∧ listp (s-expr (s))
 ∧ (car (s-expr (s)) = `'if`)
 ∧ ($c \not\simeq$ `0`)
 ∧ s-good-statep (s, c)
 ∧ (p-psw (lr-if-ok (lr-eval (**t**,
 lr-set-pos (s->lr1 (s, l, $table$), dv (s-pos (s), `1`)),
 c)))

$\neq$ 'run))
$\rightarrow$ (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)
= lr-if-ok (lr-eval (**t**,
lr-set-pos (s->lr1 (*s*, *l*, *table*), dv (s-pos (*s*), 1)),
c)))

THEOREM: lr-eval-s->lr1-temp-eval-opener
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ (car (s-expr (*s*)) = S-TEMP-EVAL)
$\wedge$ (*flag* $\neq$ 'list)
$\wedge$ (*c* $\not\simeq$ 0)
$\wedge$ listp (s-expr (*s*))
$\wedge$ s-good-statep (*s*, *c*))
$\rightarrow$ (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)
= lr-set-temp (lr-eval (**t**,
lr-set-pos (s->lr1 (*s*, *l*, *table*),
dv (s-pos (*s*), 1)),
c),
top (p-temp-stk (lr-eval (**t**,
lr-set-pos (s->lr1 (*s*,
l,
table),
dv (s-pos (*s*), 1)),
c))),
caddr (lr-expr (s->lr1 (*s*, *l*, *table*))))))

THEOREM: lr-eval-s->lr1-temp-test-opener
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ (car (s-expr (*s*)) = S-TEMP-TEST)
$\wedge$ (*flag* $\neq$ 'list)
$\wedge$ (*c* $\not\simeq$ 0)
$\wedge$ listp (s-expr (*s*))
$\wedge$ s-good-statep (*s*, *c*))
$\rightarrow$ (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)
= **if** p-max-temp-stk-size (*l*) $\not<$ (2 + length (p-temp-stk (*l*)))
then if lr-eval-temp-setp (s->lr1 (*s*, *l*, *table*))
then lr-do-temp-fetch (s->lr1 (*s*, *l*, *table*))
else lr-set-temp (lr-eval (**t**,
lr-set-pos (s->lr1 (*s*,
l,
table),
dv (s-pos (*s*), 1)),
c),
top (p-temp-stk (lr-eval (**t**,

lr-set-pos (s->lr1 (*s*,
l,
table),
dv (s-pos (*s*),
1)),
c))),
caddr (lr-expr (s->lr1 (*s*, *l*, *table*)))) **endif**
else lr-set-error (s->lr1 (*s*, *l*, *table*),
'lr-temp-setp-temp-stack-overflow) **endif**)

THEOREM: lr-eval-s->lr1-temp-fetch-opener
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ (car (s-expr (*s*)) = S-TEMP-FETCH)
∧ (*flag* ≠ 'list)
∧ (*c* ≠ 0)
∧ listp (s-expr (*s*))
∧ s-good-statep (*s*, *c*))
→ (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)
= lr-do-temp-fetch (s->lr1 (*s*, *l*, *table*)))

THEOREM: lr-eval-s->lr1-quote-opener
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ (car (s-expr (*s*)) = 'quote)
∧ (*flag* ≠ 'list)
∧ (*c* ≠ 0)
∧ listp (s-expr (*s*))
∧ s-good-statep (*s*, *c*))
→ (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)
= lr-push-tstk (s->lr1 (*s*, *l*, *table*),
cadr (lr-expr (s->lr1 (*s*, *l*, *table*)))))

THEOREM: lr-params-lr-set-temp
lr-params (*frame*, lr-set-temp (*l*, *value*, *var-name*)) = lr-params (*frame*, *l*)

THEOREM: lr-temps-lr-set-temp
lr-temps (*frame*, lr-set-temp (*l*, *value*, *var-name*)) = lr-temps (*frame*, *l*)

THEOREM: firstn-put-assoc
firstn (*n*, put-assoc (*val*, *name*, *alist*)) = put-assoc (*val*, *name*, firstn (*n*, *alist*))

THEOREM: strip-cars-nil-fact
(**nil** = strip-cars (*y*)) = (¬ listp (*y*))

DEFINITION:
induct-hint-13 (*e*, *x*, *y*)

```
=   if listp(x)
    then if listp(y)
         then if e = caar(x) then t
              else induct-hint-13(e, cdr(x), cdr(y)) endif
         else t endif
    else t endif
```

THEOREM: strip-cars-equal-definedp-equal
(strip-cars(x) = strip-cars(y)) → (definedp(e, x) = definedp(e, y))

THEOREM: lr-eval-preserves-definedp-firstn-bindings-car-p-ctrl-stk
```
(proper-p-statep(lr->p(l))
 ∧  good-posp(flag, offset(p-pc(l)), program-body(p-current-program(l)))
 ∧  lr-programs-properp(l, table)
 ∧  (p-psw(lr-eval(flag, l, c)) = 'run))
→  (definedp(x, firstn(n, bindings(car(p-ctrl-stk(lr-eval(flag, l, c)))))))
     =  definedp(x, firstn(n, bindings(car(p-ctrl-stk(l))))))
```

DEFINITION:
```
disjointp(list1, list2)
=   if listp(list1)
    then (car(list1) ∉ list2) ∧ disjointp(cdr(list1), list2)
    else t endif
```

THEOREM: member-disjointp-non-member-1
(disjointp(x, y) ∧ (e ∈ x)) → (e ∉ y)

THEOREM: lr-eval-preserves-definedp-fn-bindings-car-ctrl-stk-set-pos
```
(proper-p-statep(lr->p(l))
 ∧  good-posp1(pos, program-body(p-current-program(l)))
 ∧  lr-programs-properp(l, table)
 ∧  (p-psw(lr-eval(t, lr-set-pos(l, pos), c)) = 'run))
→  (definedp(x,
               firstn(n,
                      bindings(car(p-ctrl-stk(lr-eval(t,
                                                      lr-set-pos(l, pos),
                                                      c))))))
     =  definedp(x, firstn(n, bindings(car(p-ctrl-stk(l))))))
```

THEOREM: lr-params-p-frame-not-definedp-put-assoc-anything
```
(proper-p-statep(lr->p(l))
 ∧  good-posp1(offset(p-pc(l)), program-body(p-current-program(l)))
 ∧  good-posp1(pos, program-body(p-current-program(l)))
 ∧  lr-programs-properp(l, table)
 ∧  disjointp(formal-vars(p-current-program(l)),
```

```
            strip-cars (temp-var-dcls (p-current-program (l))))
 ∧  listp (lr-expr (l))
 ∧  ((car (lr-expr (l)) = S-TEMP-EVAL)
     ∨  (car (lr-expr (l)) = S-TEMP-TEST))
 ∧  (p-psw (lr-eval (t, lr-set-pos (l, pos), c)) = 'run))
 →  (lr-params (p-frame (put-assoc (anything,
                                    caddr (lr-expr (l)),
                                    bindings (car (p-ctrl-stk (lr-eval (t,
                                                                        lr-set-pos (l,
                                                                                    pos),
                                                                        c))))),
                        ret-pc),
                l)
    =  lr-params (car (p-ctrl-stk (lr-eval (t, lr-set-pos (l, pos), c))), l))
```

DEFINITION:

```
induct-hint-14 (s-temps, lr-temps, temp-alist)
=  if listp (s-temps)
   then if listp (lr-temps)
        then if listp (temp-alist)
             then if cdar (lr-temps) = LR-UNDEF-ADDR
                  then induct-hint-14 (cdr (s-temps),
                                       cdr (lr-temps),
                                       cdr (temp-alist))
                  else induct-hint-14 (cdr (s-temps),
                                       cdr (lr-temps),
                                       cdr (temp-alist)) endif
             else t endif
        else t endif
   else t endif
```

THEOREM: put-assoc-opener-1

```
((name ≠ caar (alist)) ∧ listp (alist))
→  (put-assoc (val, name, alist)
    =  cons (car (alist), put-assoc (val, name, cdr (alist))))
```

THEOREM: put-assoc-opener-2

```
(listp (alist3)
 ∧  (caar (alist3) ∉ strip-cars (cdr (alist3)))
 ∧  definedp (s-expr, alist1)
 ∧  (strip-cars (alist1) = strip-cars (alist2))
 ∧  (strip-cdrs (alist2) = strip-cars (cdr (alist3))))
→  (put-assoc (val, cdr (assoc (s-expr, alist2)), alist3)
    =  cons (car (alist3),
             put-assoc (val, cdr (assoc (s-expr, alist2)), cdr (alist3))))
```

THEOREM: not-lr-valp-lr-undef-addr
(lr-proper-heapp (*data-seg*) ∧ lr-proper-p-areasp (*data-seg*))
→ (¬ lr-valp (*value*, identity (LR-UNDEF-ADDR), *data-seg*))

THEOREM: lr-s-similar-temps-put-assoc-put-assoc-helper-1
(listp (*s-temps*)
∧ listp (*lr-temps*)
∧ lr-s-similar-temps (*s-temps*, *lr-temps*, *data-seg*)
∧ lr-valp (*value*, *addr*, *data-seg*)
∧ lr-proper-heapp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ (*name1* = caar (*s-temps*))
∧ (*name2* = caar (*lr-temps*)))
→ lr-s-similar-temps (put-assoc (list (**t**, *value*), *name1*, *s-temps*),
put-assoc (*addr*, *name2*, *lr-temps*),
data-seg)

THEOREM: lr-s-similar-temps-put-assoc-put-assoc-helper
(lr-s-similar-temps (*s-temps*, *lr-temps*, *data-seg*)
∧ lr-valp (*value*, *addr*, *data-seg*)
∧ lr-proper-heapp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ (strip-cars (*temp-alist*) = strip-cars (*s-temps*))
∧ (strip-cdrs (*temp-alist*) = strip-cars (*lr-temps*))
∧ no-duplicatesp (strip-cars (*lr-temps*))
∧ definedp (*s-expr*, *s-temps*))
→ lr-s-similar-temps (put-assoc (list (**t**, *value*), *s-expr*, *s-temps*),
put-assoc (*addr*,
cdr (assoc (*s-expr*, *temp-alist*)),
lr-temps),
data-seg)

THEOREM: disjointp-cons-arg2
(disjointp (*list1*, *list2*) ∧ (*x* ∉ *list1*))
→ disjointp (*list1*, cons (*x*, *list2*))

THEOREM: disjointp-nlistp-arg2
(*list2* ≃ **nil**) → disjointp (*list1*, *list2*)

THEOREM: disjointp-lr-make-temp-name-alist-1
disjointp (*formals*,
strip-cdrs (lr-make-temp-name-alist-1 (*initial*,
num-list,
temp-list,
formals)))

THEOREM: lr-s-similar-statesp-s-change-temp-helper-2
let *lr-eval* **be** lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)
in
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ good-posp1 (*pos*, s-body (s-prog (*s*)))
∧ lr-s-similar-statesp (*s-params*, s-temps (*s-eval*), *lr-eval*, *table*)
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ s-good-statep (*s*, *c*)
∧ listp (s-expr (*s*))
∧ ((car (s-expr (*s*)) = S-TEMP-EVAL)
∨ (car (s-expr (*s*)) = S-TEMP-TEST))
∧ (p-psw (*lr-eval*) = 'run)
∧ (*lr-expr* = caddr (lr-expr (s->lr1 (*s*, *l*, *table*)))))
→ (lr-s-similar-statesp (*s-params*,
s-temps (s-change-temp (*s-eval*,
s-expr,
value)),
lr-set-temp (*lr-eval*, *addr*, *lr-expr*),
table)
= lr-s-similar-temps (put-assoc (list (**t**, *value*),
s-expr,
s-temps (*s-eval*)),
lr-temps (p-frame (put-assoc (*addr*,
lr-expr,
bindings (car (p-ctrl-stk (*lr-eval*)))),
ret-pc (car (p-ctrl-stk (*lr-eval*)))),
s->lr1 (*s*, *l*, *table*)),
p-data-segment (*lr-eval*))) **endlet**

THEOREM: good-posp1-dv-1-temps-lr-expr
(((car (s-expr (*s*)) = S-TEMP-EVAL) ∨ (car (s-expr (*s*)) = S-TEMP-TEST))
∧ listp (s-expr (*s*))
∧ good-posp1 (s-pos (*s*), s-body (assoc (s-pname (*s*), s-progs (*s*)))))
→ good-posp1 (dv (s-pos (*s*), 1), s-body (assoc (s-pname (*s*), s-progs (*s*))))

THEOREM: put-assoc-restn
(¬ definedp (*name*, firstn (*n*, *alist*)))
→ (put-assoc (*val*, *name*, restn (*n*, *alist*))
= restn (*n*, put-assoc (*val*, *name*, *alist*)))

THEOREM: disjointp-plist-arg-2
disjointp (*x*, plist (*y*)) = disjointp (*x*, *y*)

THEOREM: not-disjointp-member-arg1-cons-arg2
(*v* ∈ *y*) → (¬ disjointp (*y*, cons (*v*, *z*)))

THEOREM: member-disjointp-cons-arg2
$(v \notin y) \rightarrow (\text{disjointp}(y, \text{cons}(v, z)) = \text{disjointp}(y, z))$

THEOREM: disjointp-commutative
$\text{disjointp}(x, y) = \text{disjointp}(y, x)$

THEOREM: disjointp-lr-make-temp-name-alist-2
$\text{disjointp}(\text{strip-cdrs}(\text{lr-make-temp-name-alist-1}(\textit{initial}, \textit{num-list}, \textit{temp-list}, \textit{formals})), \textit{formals})$

THEOREM: proper-p-statep-lr->p-s->lr1-strip-cars-bindings-ctrl-stk
$(\text{proper-p-statep}(\text{lr->p}(\text{s->lr1}(s, l, \textit{table}))))$
$\wedge \quad \text{definedp}(\text{s-pname}(s), \text{s-progs}(s)))$
$\rightarrow \quad (\text{strip-cars}(\text{bindings}(\text{car}(\text{p-ctrl-stk}(l)))))$
$= \quad \text{append}(\text{s-formals}(\text{assoc}(\text{s-pname}(s), \text{s-progs}(s))),$
$\text{strip-cdrs}(\text{lr-make-temp-name-alist}(\text{s-temp-list}(\text{assoc}(\text{s-pname}(s), \text{s-progs}(s))),$
$\text{s-formals}(\text{assoc}(\text{s-pname}(s), \text{s-progs}(s))))))$

THEOREM: lr-programs-properp-lr->p-s->lr1-definedp-s-pname
$(\neg\ \text{definedp}(\text{s-pname}(s), \text{s-progs}(s)))$
$\rightarrow \quad (\neg\ \text{lr-programs-properp}(\text{s->lr1}(s, l, \textit{table}), \textit{table}))$

THEOREM: lr-temps-p-frame-put-assoc
$(\text{proper-p-statep}(\text{lr->p}(\text{s->lr1}(s, l, \textit{table}))))$
$\wedge \quad \text{lr-programs-properp}(\text{s->lr1}(s, l, \textit{table}), \textit{table})$
$\wedge \quad \text{definedp}(\text{s-pname}(s), \text{s-progs}(s))$
$\wedge \quad \text{listp}(\text{s-expr}(s))$
$\wedge \quad ((\text{car}(\text{s-expr}(s)) = \text{S-TEMP-EVAL})$
$\vee \quad (\text{car}(\text{s-expr}(s)) = \text{S-TEMP-TEST}))$
$\wedge \quad \text{good-posp1}(\text{s-pos}(s), \text{s-body}(\text{s-prog}(s)))$
$\wedge \quad (\text{p-psw}(\text{lr-eval}(\mathtt{t}, \textit{l2}, c)) = \mathtt{'run})$
$\wedge \quad (\textit{lr-expr} = \text{caddr}(\text{lr-expr}(\text{s->lr1}(s, l, \textit{table}))))$
$\wedge \quad (\textit{l2} = \text{lr-set-pos}(\text{s->lr1}(s, l, \textit{table}), \text{dv}(\text{s-pos}(s), \mathtt{1})))$
$\rightarrow \quad (\text{lr-temps}(\text{p-frame}(\text{put-assoc}(\textit{val}, \textit{lr-expr}, \text{bindings}(\text{car}(\text{p-ctrl-stk}(\text{lr-eval}(\mathtt{t}, \textit{l2}, c))))),$
$\textit{ret-pc}),$
$\text{s->lr1}(s, l, \textit{table}))$
$= \quad \text{put-assoc}(\textit{val},$
$\text{caddr}(\text{lr-expr}(\text{s->lr1}(s, l, \textit{table}))),$

lr-temps (car (p-ctrl-stk (lr-eval (**t**, *l2*, *c*))),
s->lr1 (*s*, *l*, *table*))))

THEOREM: strip-cars-lr-temps-strip-cars-temp-var-dcls
(s-good-statep (*s*, *c*)
∧ proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ (*frame* = top (p-ctrl-stk (s->lr1 (*s*, *l*, *table*)))))
→ (strip-cars (lr-temps (*frame*, s->lr1 (*s*, *l*, *table*)))
= strip-cdrs (lr-make-temp-name-alist (s-temp-list (assoc (s-pname (*s*),
s-progs (*s*))),
s-formals (assoc (s-pname (*s*),
s-progs (*s*))))))

EVENT: Disable proper-p-statep-lr->p-s->lr1-strip-cars-bindings-ctrl-stk.

THEOREM: lr-s-similar-statesp-s-change-temp-helper-1
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ listp (s-expr (*s*))
∧ ((car (s-expr (*s*)) = S-TEMP-EVAL)
∨ (car (s-expr (*s*)) = S-TEMP-TEST)
∨ (car (s-expr (*s*)) = S-TEMP-FETCH)))
→ (caddr (lr-expr (s->lr1 (*s*, *l*, *table*)))
= cdr (assoc (cadr (s-expr (*s*)),
lr-make-temp-name-alist (s-temp-list (s-prog (*s*)),
s-formals (s-prog (*s*))))))

THEOREM: lr-s-similar-statesp-s->lr1-lr-similar-temps
lr-s-similar-statesp (*s-params*, *s-temps*, *l*, *table*)
→ lr-s-similar-temps (*s-temps*,
lr-temps (top (p-ctrl-stk (*l*)), *l*),
p-data-segment (*l*))

THEOREM: count-codelist1-cons
count-codelist1 (cons (*x*, *y*)) = (*x* + (10 ∗ count-codelist1 (*y*)))

THEOREM: equal-append-initial
(append (*x*, *y*) = append (*x*, *z*)) = (*y* = *z*)

THEOREM: plist-listp-x-append-x-not-0
plistp (*x*) → ((append (*x*, 0) = 0) = (*x* = **nil**))

THEOREM: equal-append-final-0
(append (*y*, 0) = append (*z*, 0)) = (plist (*y*) = plist (*z*))

THEOREM: count-codelist1-append-non-listp
$(\neg\ \text{listp}\,(z))$
$\rightarrow\quad (\text{count-codelist1}\,(\text{append}\,(\textit{num-list},\ z)) = \text{count-codelist1}\,(\textit{num-list}))$

THEOREM: not-equal-make-symbol-car-gensym
$(\text{count-codelist1}\,(\textit{num-list1}) < \text{count-codelist1}\,(\textit{num-list2}))$
$\rightarrow\quad (\text{make-symbol}\,(\textit{initial},\ \textit{num-list1})$
$\neq\quad \text{car}\,(\text{gensym}\,(\textit{initial},\ \textit{num-list2},\ \textit{atom-list})))$

THEOREM: count-codelist1-cdr-gensym
$(\text{count-codelist1}\,(\textit{num-list1}) < \text{count-codelist1}\,(\textit{num-list2}))$
$\rightarrow\quad (\text{count-codelist1}\,(\textit{num-list1})$
$<\quad \text{count-codelist1}\,(\text{cdr}\,(\text{gensym}\,(\textit{initial},\ \textit{num-list2},\ \textit{atom-list}))))$

THEOREM: not-member-make-symbol-lr-make-temp-name-alist-1-incr
$(\text{count-codelist1}\,(\textit{num-list1}) < \text{count-codelist1}\,(\textit{num-list2}))$
$\rightarrow\quad ((\text{make-symbol}\,(\textit{initial},\ \textit{num-list1})$
$\in\quad \text{strip-cdrs}\,(\text{lr-make-temp-name-alist-1}\,(\textit{initial},$
$\textit{num-list2},$
$\textit{temp-list},$
$\textit{formals})))$
$=\quad \mathbf{f})$

THEOREM: not-member-car-gensym-lr-make-temp-name-alist-1-cdr
$(\text{car}\,(\text{gensym}\,(\textit{initial},\ \textit{num-list},\ \textit{atoms}))$
$\in\quad \text{strip-cdrs}\,(\text{lr-make-temp-name-alist-1}\,(\textit{initial},$
$\text{cdr}\,(\text{gensym}\,(\textit{initial},$
$\textit{num-list},$
$\textit{atoms})),$
$\textit{temp-list},$
$\textit{formals})))$
$=\quad \mathbf{f}$

THEOREM: no-duplicatesp-strip-cdrs-lr-make-temp-name-alist-1
$\text{no-duplicatesp}\,(\text{strip-cdrs}\,(\text{lr-make-temp-name-alist-1}\,(\textit{initial},$
$\textit{num-list},$
$\textit{temp-list},$
$\textit{formals})))$

THEOREM: no-duplicatesp-strip-cdrs-lr-make-temp-name-alist
$\text{no-duplicatesp}\,(\text{strip-cdrs}\,(\text{lr-make-temp-name-alist}\,(\textit{temp-list},\ \textit{formals})))$

THEOREM: definedp-s-temps-s-eval
$(\text{s-err-flag}\,(\text{s-eval}\,(\textit{flag},\ s,\ c)) = \texttt{'run})$
$\rightarrow\quad (\text{definedp}\,(x,\ \text{s-temps}\,(\text{s-eval}\,(\textit{flag},\ s,\ c))) = \text{definedp}\,(x,\ \text{s-temps}\,(s)))$

THEOREM: strip-cars-lr-make-temp-name-alist-1
strip-cars (lr-make-temp-name-alist-1 (*initial*, *num-list*, *temp-list*, *formals*))
= plist (*temp-list*)

THEOREM: strip-cars-lr-make-temp-name-alist
strip-cars (lr-make-temp-name-alist (*temp-list*, *formals*)) = plist (*temp-list*)

THEOREM: lr-eval-preserves-strip-cars-lr-temps-car-p-ctrl-stk
(proper-p-statep (lr->p (*l*))
∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = 'run))
→ (strip-cars (lr-temps (car (p-ctrl-stk (lr-eval (*flag*, *l*, *c*))), *l2*))
= strip-cars (lr-temps (car (p-ctrl-stk (*l*)), *l2*)))

THEOREM: lr-s-similar-statesp-s-change-temp
let *s-l* **be** s->lr1 (*s*, *l*, *table*),
lr-eval **be** lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)
in
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ lr-s-similar-statesp (*s-params*, s-temps (*s-eval*), *lr-eval*, *table*)
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ s-good-statep (*s*, *c*)
∧ listp (p-temp-stk (*lr-eval*))
∧ lr-check-result (**t**,
s-ans (*s-eval*),
p-temp-stk (*lr-eval*),
p-data-segment (*lr-eval*),
orig-temp-stk)
∧ listp (s-expr (*s*))
∧ ((car (s-expr (*s*)) = S-TEMP-EVAL)
∨ (car (s-expr (*s*)) = S-TEMP-TEST))
∧ (p-psw (*lr-eval*) = 'run)
∧ (s-err-flag (*s-eval*) = 'run)
∧ (*s-eval* = s-eval (**t**, s-set-pos (*s*, *pos*), *c*))
∧ (*value* = caddr (lr-expr (*s-l*)))
∧ (*pos* = dv (s-pos (*s*), 1)))
→ lr-s-similar-statesp (*s-params*,
s-temps (s-change-temp (*s-eval*,
cadr (s-expr (*s*)),
s-ans (*s-eval*))),
lr-set-temp (*lr-eval*,
car (p-temp-stk (*lr-eval*)),
value),

table) **endlet**

EVENT: Disable lr-s-similar-statesp-s-change-temp-helper-2.

THEOREM: lr-temps-lr-do-temp-fetch
lr-temps (*frame*, lr-do-temp-fetch (*l*)) = lr-temps (*frame*, *l*)

THEOREM: lr-params-lr-do-temp-fetch
lr-params (*frame*, lr-do-temp-fetch (*l*)) = lr-params (*frame*, *l*)

THEOREM: lr-s-simlar-statesp-lr-do-temp-fetch
lr-s-similar-statesp (*s-params*, *s-temps*, lr-do-temp-fetch (*l*), *table*)
= lr-s-similar-statesp (*s-params*, *s-temps*, *l*, *table*)

THEOREM: not-member-no-duplicates-cdr-assoc-helper
(no-duplicatesp (*list*)
∧ (strip-cdrs (*alist*) = *list*)
∧ (*name* ∉ *list*)
∧ definedp (*s-expr*, *alist*))
→ (cdr (assoc (*s-expr*, *alist*)) ≠ *name*)

THEOREM: not-member-no-duplicates-cdr-assoc
(no-duplicatesp (*list*)
∧ (strip-cdrs (*alist1*) = *list*)
∧ (*name* ∉ *list*)
∧ definedp (*s-expr*, *alist2*)
∧ (strip-cars (*alist2*) = strip-cars (*alist1*)))
→ (cdr (assoc (*s-expr*, *alist1*)) ≠ *name*)

THEOREM: not-equal-lr-s-eval-temp-setp-not-lr-s-similar-temps
((*lr-expr* = cdr (assoc (*s-expr*, *temp-alist*)))
∧ lr-proper-heapp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ lr-s-similar-temps (*s-temps*, *lr-temps*, *data-seg*)
∧ (strip-cars (*temp-alist*) = strip-cars (*s-temps*))
∧ (strip-cdrs (*temp-alist*) = strip-cars (*lr-temps*))
∧ no-duplicatesp (strip-cars (*lr-temps*))
∧ definedp (*s-expr*, *s-temps*))
→ ((cdr (assoc (*lr-expr*, *lr-temps*)) ≠ LR-UNDEF-ADDR)
↔ cadr (assoc (*s-expr*, *s-temps*)))

THEOREM: definedp-strip-cars-append-member-x-2
(strip-cars (*x*) = append (*y*, *z*))
→ ((*e* ∈ *z*) = definedp (*e*, restn (length (*y*), *x*)))

THEOREM: not-iff-lr-s-temp-setp-not-lr-s-similar-statesp-helper
((x ∈ strip-cdrs (lr-make-temp-name-alist (s-temp-list (assoc (s-pname (s),
s-progs (s))),
s-formals (assoc (s-pname (s),
s-progs (s))))))
∧ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
∧ definedp (s-pname (s), s-progs (s)))
→ (cdr (assoc (x, lr-temps (car (p-ctrl-stk (l)), s->lr1 (s, l, *table*))))
= cdr (assoc (x, bindings (car (p-ctrl-stk (l))))))

THEOREM: lr-programs-properp-member-lr-expr-temps
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
∧ lr-programs-properp (l, *table*)
∧ ((car (lr-expr (l)) = S-TEMP-FETCH)
∨ (car (lr-expr (l)) = S-TEMP-EVAL)
∨ (car (lr-expr (l)) = S-TEMP-TEST)))
→ (caddr (lr-expr (l))
∈ strip-cars (temp-var-dcls (p-current-program (l))))

THEOREM: not-iff-lr-s-temp-setp-not-lr-s-similar-statesp
(proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
∧ lr-proper-heapp (p-data-segment (l))
∧ good-posp1 (s-pos (s), s-body (s-prog (s)))
∧ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
∧ listp (s-expr (s))
∧ ((car (s-expr (s)) = S-TEMP-TEST)
∨ (car (s-expr (s)) = S-TEMP-FETCH))
∧ (¬ (lr-eval-temp-setp (s->lr1 (s, l, *table*))
↔ s-temp-setp (cadr (s-expr (s)), s-temps (s))))
∧ s-good-statep (s, c))
→ (¬ lr-s-similar-statesp (*s-params*, s-temps (s), s->lr1 (s, l, *table*), *table*))

THEOREM: lr-valp-lr-s-eval-lr-s-similar-temps
((*lr-expr* = cdr (assoc (*s-expr*, *temp-alist*)))
∧ lr-proper-heapp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ lr-s-similar-temps (*s-temps*, *lr-temps*, *data-seg*)
∧ (strip-cars (*temp-alist*) = strip-cars (*s-temps*))
∧ (strip-cdrs (*temp-alist*) = strip-cars (*lr-temps*))
∧ no-duplicatesp (strip-cars (*lr-temps*))
∧ definedp (*s-expr*, *s-temps*)
∧ (cdr (assoc (*lr-expr*, *lr-temps*)) ≠ LR-UNDEF-ADDR))
→ lr-valp (caddr (assoc (*s-expr*, *s-temps*)),
cdr (assoc (*lr-expr*, *lr-temps*)),
data-seg)

THEOREM: member-cdr-assoc-strip-cdrs-definedp
definedp (*x*, *alist*) → (cdr (assoc (*x*, *alist*)) ∈ strip-cdrs (*alist*))

THEOREM: definedp-pairlist
definedp (*x*, pairlist (*temp-list*, *anything*)) = (*x* ∈ *temp-list*)

THEOREM: definedp-lr-make-temp-name-alist-1
definedp (*x*, lr-make-temp-name-alist-1 (*initial*, *num-list*, *temp-list*, *formals*))
= (*x* ∈ *temp-list*)

THEOREM: definedp-lr-make-temp-name-alist
definedp (*x*, lr-make-temp-name-alist (*temp-list*, *formals*))
= (*x* ∈ *temp-list*)

THEOREM: p-temp-stk-lr-do-temp-fetch-p-psw-run
(p-psw (lr-do-temp-fetch (*l*)) = 'run)
→ (p-temp-stk (lr-do-temp-fetch (*l*))
= push (local-var-value (caddr (lr-expr (*l*)), p-ctrl-stk (*l*)),
p-temp-stk (*l*)))

THEOREM: lr-check-result-lr-do-temp-fetch
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ s-good-statep (*s*, *c*)
∧ (*c* ≠ 0)
∧ listp (s-expr (*s*))
∧ ((car (s-expr (*s*)) = S-TEMP-TEST)
∨ (car (s-expr (*s*)) = S-TEMP-FETCH))
∧ lr-proper-heapp (p-data-segment (*l*))
∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
∧ lr-eval-temp-setp (s->lr1 (*s*, *l*, *table*))
∧ (*value* = caddr (lr-expr (s->lr1 (*s*, *l*, *table*)))))
→ lr-check-result (t,
caddr (assoc (cadr (s-expr (*s*)), s-temps (*s*))),
cons (cdr (assoc (*value*, bindings (car (p-ctrl-stk (*l*))))),
p-temp-stk (*l*)),
p-data-segment (*l*),
p-temp-stk (*l*))

THEOREM: lr-do-temp-fetch-run-lr-eval-temp-setp
(p-psw (lr-do-temp-fetch (*l*)) = 'run) → lr-eval-temp-setp (*l*)

THEOREM: lr-s-similar-const-table-lr-valp-assoc
(definedp (*value*, *table*) ∧ lr-s-similar-const-table (*table*, *data-seg*))
→ lr-valp (*value*, cdr (assoc (*value*, *table*)), *data-seg*)

THEOREM: lr-proper-exprp-list-quote-opener
(*flag* ≠ 'list)
→ (lr-proper-exprp (*flag*,
list ('quote, *addr*),
program-names,
formals,
temps,
table)
= ((type (*addr*) = 'addr) ∧ (*addr* ∈ strip-cdrs (*table*))))

THEOREM: lr-check-result-lr-push-tstk-quote
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) = 'quote)
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
∧ s-good-statep (*s*, *c*)
∧ lr-proper-heapp (p-data-segment (*l*))
∧ (p-psw (lr-push-tstk (s->lr1 (*s*, *l*, *table*),
cadr (lr-expr (s->lr1 (*s*, *l*, *table*)))))
= 'run))
→ lr-check-result (**t**,
cadr (s-expr (*s*)),
p-temp-stk (lr-push-tstk (s->lr1 (*s*, *l*, *table*),
cadr (lr-expr (s->lr1 (*s*, *l*, *table*))))),
p-data-segment (*l*),
p-temp-stk (*l*))

THEOREM: lr-eval-subrp-user-funcall-opener
let *lr-eval-list* **be** lr-eval ('list,
lr-set-pos (s->lr1 (*s*, *l*, *table*), dv (s-pos (*s*), 1)),
c)
in
((*flag* ≠ 'list)
∧ (*c* ≠ 0)
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (car (s-expr (*s*)) ≠ S-TEMP-EVAL)
∧ (car (s-expr (*s*)) ≠ S-TEMP-TEST)
∧ (car (s-expr (*s*)) ≠ S-TEMP-FETCH)
∧ (car (s-expr (*s*)) ≠ 'quote)
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ s-good-statep (*s*, *c*))
→ (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)

```
=  if p-psw (lr-eval-list) ≠ 'run then lr-eval-list
   elseif subrp (car (s-expr (s)))
   then lr-apply-subr (s->lr1 (s, l, table), lr-eval-list)
   elseif litatom (car (s-expr (s)))
   then lr-set-expr (lr-pop-cstk (lr-eval (t,
                                           lr-funcall (s->lr1 (s,
                                                               l,
                                                               table),
                                                       lr-eval-list),
                                           c − 1)),
                                 s->lr1 (s, l, table),
                                 s-pos (s))
   else lr-set-error (s->lr1 (s, l, table),
                      'bad-instruction) endif) endlet
```

THEOREM: length-cdr-lr-expr-funcall
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ listp (lr-expr (*l*))
 ∧ (subrp (car (lr-expr (*l*))) ∨ litatom (car (lr-expr (*l*)))))
→ (length (cdr (lr-expr (*l*))) = arity (car (lr-expr (*l*)))))

DEFINITION:
induct-hint-8 (*n*, *value*, *temp-stk*)
= **if** *n* ≃ 0 **then t**
  else induct-hint-8 (*n* − 1, cdr (*value*), cdr (*temp-stk*)) **endif**

THEOREM: lr-check-result1-lr-valp-get-n-lessp-length
(lr-check-result1 (*values*, *temp-stk*, *data-seg*) ∧ (*n* < length (*values*)))
→ lr-valp (get (*n*, *values*), get (*n*, *temp-stk*), *data-seg*)

THEOREM: lr-valp-lr-good-pointerp
lr-valp (*value*, *addr*, *data-seg*) → lr-good-pointerp (*addr*, *data-seg*)

THEOREM: lr-check-result1-lr-good-pointerp-get-n-lessp-car
(lr-check-result1 (*values*, *temp-stk*, *data-seg*) ∧ (length (*values*) ≮ 1))
→ ((type (car (*temp-stk*)) = 'addr)
   ∧ (cddr (car (*temp-stk*)) = **nil**)
   ∧ listp (car (*temp-stk*))
   ∧ adpp (untag (car (*temp-stk*)), *data-seg*)
   ∧ lr-boundary-nodep (car (*temp-stk*))
   ∧ (area-name (car (*temp-stk*)) = identity (LR-HEAP-NAME))
   ∧ (type (fetch (add-addr (car (*temp-stk*),
                             identity (LR-REF-COUNT-OFFSET)),
                   data-seg))
      = 'nat))

THEOREM: lr-check-result1-lr-good-pointerp-get-n-lessp-cadr
(lr-check-result1 (*values*, *temp-stk*, *data-seg*) ∧ (length (*values*) ≮ 2))
→ ((type (cadr (*temp-stk*)) = 'addr)
∧ (cddr (cadr (*temp-stk*)) = **nil**)
∧ listp (cadr (*temp-stk*))
∧ adpp (untag (cadr (*temp-stk*)), *data-seg*)
∧ lr-boundary-nodep (cadr (*temp-stk*))
∧ (area-name (cadr (*temp-stk*)) = identity (LR-HEAP-NAME))
∧ (type (fetch (add-addr (cadr (*temp-stk*),
identity (LR-REF-COUNT-OFFSET)),
data-seg))
= 'nat))

THEOREM: p-run-subr-preserves-lr-proper-heapp2
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ lr-programs-properp (*new-l*, *table*)
∧ listp (lr-expr (*l*))
∧ proper-p-statep (lr->p (*new-l*))
∧ lr-proper-free-listp (p-data-segment (*new-l*))
∧ adpp (untag (lr-max-node (p-data-segment (*new-l*))), p-data-segment (*new-l*))
∧ lr-boundary-nodep (lr-max-node (p-data-segment (*new-l*)))
∧ (p-psw (*new-l*) = 'run)
∧ (p-psw (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*), lr-return-pc (*l*))))
= 'run)
∧ lr-proper-heapp2 (*addr*, p-data-segment (*new-l*))
∧ lr-nodep (*addr*, p-data-segment (*new-l*))
∧ (p-prog-segment (*l*) = p-prog-segment (*new-l*))
∧ lr-check-result ('list,
value,
p-temp-stk (*new-l*),
p-data-segment (*new-l*),
p-temp-stk (*l*))
∧ (length (*value*) = length (cdr (lr-expr (*l*)))))
→ lr-proper-heapp2 (*addr*,
p-data-segment (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*),
lr-return-pc (*l*)))))

THEOREM: lr-apply-subr-preserves-lr-proper-heapp2
let *new-l* **be** lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)
in
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))

$\wedge$ good-posp ('list, *pos*, program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ listp (lr-expr (*l*))
$\wedge$ subrp (car (lr-expr (*l*)))
$\wedge$ (car (lr-expr (*l*)) $\neq$ 'if)
$\wedge$ proper-p-statep (lr->p (*l*))
$\wedge$ lr-proper-free-listp (p-data-segment (*l*))
$\wedge$ lr-proper-heapp2 (*addr*, p-data-segment (*new-l*))
$\wedge$ adpp (untag (lr-max-node (p-data-segment (*l*))), p-data-segment (*l*))
$\wedge$ lr-boundary-nodep (lr-max-node (p-data-segment (*l*)))
$\wedge$ lr-nodep (*addr*, p-data-segment (*l*))
$\wedge$ lr-check-result ('list,
value,
p-temp-stk (*new-l*),
p-data-segment (*new-l*),
p-temp-stk (*l*))
$\wedge$ (length (*value*) = length (cdr (lr-expr (*l*))))
$\wedge$ (p-psw (*new-l*) = 'run)
$\wedge$ (p-psw (lr-apply-subr (*l*, *new-l*)) = 'run))
$\rightarrow$ lr-proper-heapp2 (*addr*, p-data-segment (lr-apply-subr (*l*, *new-l*))) **endlet**

DEFINITION:
induct-hint-15 (*s*, *c*)
= **if** listp (s-pos (*s*))
then if listp (s-expr-list (*s*))
then induct-hint-15 (s-set-expr (s-eval (**t**, *s*, *c*), *s*, nx (s-pos (*s*))), *c*)
else t endif
else t endif

THEOREM: length-s-eval-list
(listp (s-pos (*s*)) $\wedge$ (s-err-flag (s-eval ('list, *s*, *c*)) = 'run))
$\rightarrow$ (length (s-ans (s-eval ('list, *s*, *c*))) = length (s-expr-list (*s*)))

THEOREM: plistp-lr-compile-body
listp (*body*) $\rightarrow$ plistp (lr-compile-body (*flag*, *body*, *temp-alist*, *const-alist*))

THEOREM: plistp-lr-expr-s->lr1
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*))) $\wedge$ listp (s-expr (*s*)))
$\rightarrow$ plistp (lr-expr (s->lr1 (*s*, *l*, *table*)))

THEOREM: length-cdr-lr-expr-funcall-s->lr1
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
$\wedge$ listp (s-expr (*s*))
$\wedge$ (car (s-expr (*s*)) $\neq$ 'quote)

$\wedge$ (subrp (car (s-expr (s))) $\vee$ litatom (car (s-expr (s)))))
$\rightarrow$ (length (cdr (lr-expr (s->lr1 (s, l, *table*)))) = length (cdr (s-expr (s))))

THEOREM: adpp-same-signature-lr-apply-subr
same-signature (p-data-segment (*new-l*), p-data-segment (lr-apply-subr (l, *new-l*)))
$\rightarrow$ (adpp (*adp*, p-data-segment (lr-apply-subr (l, *new-l*)))
= adpp (*adp*, p-data-segment (*new-l*)))

THEOREM: lr-apply-subr-preserves-lr-proper-heapp
let *new-l* **be** lr-eval (`'list`, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c)
in
(good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ definedp (s-pname (s), s-progs (s))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-TEST)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-FETCH)
$\wedge$ (car (s-expr (s)) $\neq$ `'quote`)
$\wedge$ subrp (car (s-expr (s)))
$\wedge$ (car (s-expr (s)) $\neq$ `'if`)
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ lr-check-result (`'list`,
s-ans (s-eval (`'list`, s-set-pos (s, *pos*), c)),
p-temp-stk (*new-l*),
p-data-segment (*new-l*),
p-temp-stk (l))
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ (p-psw (lr-apply-subr (s->lr1 (s, l, *table*), *new-l*)) = `'run`)
$\wedge$ lr-proper-heapp (p-data-segment (*new-l*))
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ (s-err-flag (s-eval (`'list`, s-set-pos (s, *pos*), c)) = `'run`)
$\wedge$ (*pos* = dv (s-pos (s), `1`)))
$\rightarrow$ lr-proper-heapp (p-data-segment (lr-apply-subr (s->lr1 (s, l, *table*),
new-l))) **endlet**

THEOREM: lr-s-similar-params-lr-apply-subr
let *new-l* **be** lr-eval (`'list`, lr-set-pos (l, *pos*), c)
in
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ listp (lr-expr (l))
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (car (lr-expr (l)) $\neq$ `'if`)
$\wedge$ proper-p-statep (lr->p (*new-l*))

$\wedge$ lr-proper-heapp (p-data-segment (*new-l*))
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ (p-psw (lr-apply-subr (*l*, *new-l*)) = `'run`)
$\wedge$ lr-s-similar-params (*s-params*, *lr-params*, p-data-segment (*new-l*)))
$\rightarrow$ lr-s-similar-params (*s-params*,
lr-params,
p-data-segment (lr-apply-subr (*l*, *new-l*))) **endlet**

THEOREM: lr-params-lr-apply-subr
((area-name (p-pc (*new-l*)) = area-name (p-pc (*l*)))
$\wedge$ (p-prog-segment (*new-l*) = p-prog-segment (*l*)))
$\rightarrow$ (lr-params (*frame*, lr-apply-subr (*l*, *new-l*)) = lr-params (*frame*, *l*))

THEOREM: lr-temps-lr-apply-subr
((area-name (p-pc (*new-l*)) = area-name (p-pc (*l*)))
$\wedge$ (p-prog-segment (*new-l*) = p-prog-segment (*l*)))
$\rightarrow$ (lr-temps (*frame*, lr-apply-subr (*l*, *new-l*)) = lr-temps (*frame*, *l*))

THEOREM: lr-s-similar-temps-lr-apply-subr
let *new-l* **be** lr-eval (`'list`, lr-set-pos (*l*, *pos*), *c*)
in
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ listp (lr-expr (*l*))
$\wedge$ subrp (car (lr-expr (*l*)))
$\wedge$ (car (lr-expr (*l*)) $\neq$ `'if`)
$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ lr-proper-heapp (p-data-segment (*new-l*))
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ (p-psw (lr-apply-subr (*l*, *new-l*)) = `'run`)
$\wedge$ lr-s-similar-temps (*s-temps*, *lr-temps*, p-data-segment (*new-l*)))
$\rightarrow$ lr-s-similar-temps (*s-temps*,
lr-temps,
p-data-segment (lr-apply-subr (*l*, *new-l*))) **endlet**

THEOREM: lr-s-similar-const-table-lr-apply-subr
let *new-l* **be** lr-eval (`'list`, lr-set-pos (*l*, *pos*), *c*)
in
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table1*)
$\wedge$ listp (lr-expr (*l*))
$\wedge$ subrp (car (lr-expr (*l*)))
$\wedge$ (car (lr-expr (*l*)) $\neq$ `'if`)
$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ lr-proper-heapp (p-data-segment (*new-l*))

$\wedge$ (p-psw (*new-l*) = 'run)
$\wedge$ (p-psw (lr-apply-subr (*l*, *new-l*)) = 'run)
$\wedge$ lr-s-similar-const-table (*table2*, p-data-segment (*new-l*)))
$\rightarrow$ lr-s-similar-const-table (*table2*,
p-data-segment (lr-apply-subr (*l*, *new-l*))) **endlet**

THEOREM: lr-s-similar-statesp-lr-apply-subr
let *new-l* **be** lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)
in
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ listp (lr-expr (*l*))
$\wedge$ subrp (car (lr-expr (*l*)))
$\wedge$ (car (lr-expr (*l*)) $\neq$ 'if)
$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ lr-proper-heapp (p-data-segment (*new-l*))
$\wedge$ (p-psw (*new-l*) = 'run)
$\wedge$ (p-psw (lr-apply-subr (*l*, *new-l*)) = 'run)
$\wedge$ lr-s-similar-statesp (*s-params*, *s-temps*, *new-l*, *table*))
$\rightarrow$ lr-s-similar-statesp (*s-params*,
s-temps,
lr-apply-subr (*l*, *new-l*),
table) **endlet**

THEOREM: proper-p-statep-lr->p-lr-eval-list-helper
let *cur-expr* **be** cur-expr (offset (p-pc (*l*)),
program-body (p-current-program (*l*)))
in
((length (*cur-expr*) < 1)
$\wedge$ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ listp (*cur-expr*)
$\wedge$ (car (*cur-expr*) $\neq$ 'if)
$\wedge$ (car (*cur-expr*) $\neq$ 'quote)
$\wedge$ (litatom (car (*cur-expr*)) $\vee$ subrp (car (*cur-expr*)))
$\wedge$ proper-p-statep (lr->p (*l*))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ (p-psw (lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)) = 'run)
$\wedge$ (*pos* = dv (offset (p-pc (*l*)), 1)))
$\rightarrow$ proper-p-statep (lr->p (lr-eval ('list, lr-set-pos (*l*, *pos*), *c*))) **endlet**

THEOREM: proper-p-statep-lr->p-lr-eval-list
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ listp (lr-expr (*l*))
$\wedge$ (car (lr-expr (*l*)) $\neq$ 'if)
$\wedge$ (car (lr-expr (*l*)) $\neq$ 'quote)

$\wedge$ (subrp (car (lr-expr (l))) $\vee$ litatom (car (lr-expr (l))))
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval (`'list`, lr-set-pos (l, *pos*), c)) = `'run`)
$\wedge$ (*pos* = dv (offset (p-pc (l)), `1`)))
$\rightarrow$ proper-p-statep (lr->p (lr-eval (`'list`, lr-set-pos (l, *pos*), c)))

EVENT: Disable proper-p-statep-lr->p-lr-eval-list-helper.

THEOREM: not-listp-p-temp-stk-not-lr-check-result1
lr-check-result1 (*value*, *temp-stk*, *data-seg*)
$\rightarrow$ (length (*temp-stk*) $\not<$ length (*value*))

THEOREM: restn-add1-opener-alt
restn ($1 + n$, *list*)
= **if** listp (*list*) **then** restn (n, cdr (*list*))
else *list* **endif**

THEOREM: cdr-p-temp-stk-p-run-subr
let *new-l* **be** lr-eval (`'list`, lr-set-pos (l, *pos*), c)
in
(listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) $\neq$ `'if`)
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (p-temp-stk (l)
= restn (length (cdr (lr-expr (l))), p-temp-stk (*new-l*)))
$\wedge$ lr-check-result1 (*value*,
p-temp-stk (*new-l*),
p-data-segment (*new-l*))
$\wedge$ (length (*value*) = length (cdr (lr-expr (l))))
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p (*new-l*), lr-return-pc (l))))
= `'run`)
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ (*pos* = dv (offset (p-pc (l)), `1`)))
$\rightarrow$ (cdr (p-temp-stk (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p (*new-l*),
lr-return-pc (l)))))
= p-temp-stk (l)) **endlet**

EVENT: Disable restn-add1-opener-alt.

THEOREM: cdr-p-temp-stk-lr-apply-subr
let *new-l* **be** lr-eval ('`list`, lr-set-pos (*l*, *pos*), *c*)
in
(listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) ≠ '`if`)
∧ subrp (car (lr-expr (*l*)))
∧ (p-temp-stk (*l*)
= restn (length (cdr (lr-expr (*l*))), p-temp-stk (*new-l*)))
∧ lr-check-result1 (*value*,
p-temp-stk (*new-l*),
p-data-segment (*new-l*))
∧ (length (*value*) = length (cdr (lr-expr (*l*))))
∧ proper-p-statep (lr->p (*l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (*new-l*) = '`run`)
∧ (p-psw (lr-apply-subr (*l*, *new-l*)) = '`run`)
∧ (*pos* = dv (offset (p-pc (*l*)), `1`)))
→ (cdr (p-temp-stk (lr-apply-subr (*l*, *new-l*))) = p-temp-stk (*l*)) **endlet**

THEOREM: lr-check-result1-reverse-length-1-opener
(length (*values*) = `1`)
→ (lr-check-result1 (reverse (*values*), *temp-stk*, *data-seg*)
= (lr-valp (car (*values*), car (*temp-stk*), *data-seg*)
∧ lr-good-pointerp (car (*temp-stk*), *data-seg*)))

THEOREM: lr-check-result1-reverse-length-2-opener
(length (*values*) = `2`)
→ (lr-check-result1 (reverse (*values*), *temp-stk*, *data-seg*)
= (lr-valp (cadr (*values*), car (*temp-stk*), *data-seg*)
∧ lr-good-pointerp (cadr (*temp-stk*), *data-seg*)
∧ lr-valp (car (*values*), cadr (*temp-stk*), *data-seg*)
∧ lr-good-pointerp (car (*temp-stk*), *data-seg*)))

THEOREM: lr-valp-fetch-tag-cons-lr-valp-car-cdr
(lr-valp (*value*, *addr*, *data-seg*)
∧ (fetch (*addr*, *data-seg*) = tag ('`nat`, LR-CONS-TAG)))
→ (lr-valp (car (*value*),
fetch (add-addr (*addr*, identity (LR-CAR-OFFSET)), *data-seg*),
data-seg)
∧ lr-valp (cdr (*value*),
fetch (add-addr (*addr*, identity (LR-CDR-OFFSET)), *data-seg*),
data-seg))

THEOREM: lr-good-pointerp-type-tag-nat

(lr-proper-heapp (*data-seg*) ∧ lr-good-pointerp (*addr*, *data-seg*))
→ (type (fetch (*addr*, *data-seg*)) = 'nat)

THEOREM: lr-proper-heapp-lr-valp-f-helper
(lr-proper-heapp-nodep (LR-F-ADDR, *data-seg*)
∧ lr-proper-heapp-nodep (*addr*, *data-seg*)
∧ lr-nodep (*addr*, *data-seg*)
∧ lr-minimum-heapp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*))
→ (lr-valp (**f**, *addr*, *data-seg*) = (*addr* = LR-F-ADDR))

THEOREM: lr-proper-heapp-lr-valp-f
(lr-proper-heapp (*data-seg*) ∧ lr-proper-p-areasp (*data-seg*))
→ (lr-valp (**f**, *addr*, *data-seg*) = (*addr* = LR-F-ADDR))

THEOREM: lr-valp-equal-value-fact
(lr-valp (*value1*, *addr*, *data-seg*) ∧ lr-valp (*value2*, *addr*, *data-seg*))
→ (*value1* = *value2*)

THEOREM: lr-proper-heapp-lr-valp-0
lr-proper-heapp (*data-seg*)
→ (lr-valp (*value*, identity (LR-0-ADDR), *data-seg*) = (*value* = 0))

THEOREM: lr-proper-heapp-lr-valp-lr-f-addr
(lr-proper-heapp (*data-seg*) ∧ lr-proper-p-areasp (*data-seg*))
→ (lr-valp (*value*, identity (LR-F-ADDR), *data-seg*) = (*value* = **f**))

THEOREM: lr-proper-heapp-lr-valp-lr-t-addr
lr-proper-heapp (*data-seg*)
→ (lr-valp (*value*, identity (LR-T-ADDR), *data-seg*) = (*value* = **t**))

THEOREM: lr-valp-fetch-tag-not-cons-lr-valp-car-cdr-0
(proper-p-statep (lr->p (*l*))
∧ lr-proper-heapp (p-data-segment (*l*))
∧ (fetch (car (p-temp-stk (*l*)), p-data-segment (*l*))
≠ tag ('nat, LR-CONS-TAG))
∧ ((car (*value*) ≠ 0) ∨ (cdr (*value*) ≠ 0)))
→ (¬ lr-valp (*value*, car (p-temp-stk (*l*)), p-data-segment (*l*)))

THEOREM: lr-valp-not-tag-cons-not-listp
((¬ listp (*value*)) ∧ (fetch (*addr*, *data-seg*) = tag ('nat, LR-CONS-TAG)))
→ (¬ lr-valp (*value*, *addr*, *data-seg*))

THEOREM: lr-valp-fetch-tag-not-cons-lr-valp-listp
(proper-p-statep (lr->p (*l*))
∧ lr-proper-heapp (p-data-segment (*l*))

$\wedge$ (fetch (car (p-temp-stk (l)), p-data-segment (l))
$\neq$ tag ('nat, LR-CONS-TAG))
$\wedge$ listp ($value$))
$\rightarrow$ ($\neg$ lr-valp ($value$, car (p-temp-stk (l)), p-data-segment (l)))

THEOREM: lr-valp-cons
lr-valp (cons (x, y), $addr$, $data\text{-}seg$)
= **if** lr-good-pointerp ($addr$, $data\text{-}seg$)
then (untag (fetch ($addr$, $data\text{-}seg$)) = LR-CONS-TAG)
$\wedge$ lr-valp (x,
fetch (add-addr ($addr$, LR-CAR-OFFSET), $data\text{-}seg$),
$data\text{-}seg$)
$\wedge$ lr-valp (y,
fetch (add-addr ($addr$, LR-CDR-OFFSET), $data\text{-}seg$),
$data\text{-}seg$)
else f endif

THEOREM: lr-valp-deposit-a-list-cons
(lr-proper-free-listp ($data\text{-}seg$)
$\wedge$ lr-proper-p-areasp ($data\text{-}seg$)
$\wedge$ lr-nodep (lr-max-node ($data\text{-}seg$), $data\text{-}seg$)
$\wedge$ lr-minimum-heapp ($data\text{-}seg$)
$\wedge$ lr-boundary-nodep (lr-max-node ($data\text{-}seg$))
$\wedge$ lr-valp ($value$, $addr$, $data\text{-}seg$)
$\wedge$ lr-proper-p-areasp ($data\text{-}seg$)
$\wedge$ ($fp\text{-}addr$ = fetch (identity (LR-FP-ADDR), $data\text{-}seg$)))
$\rightarrow$ lr-valp ($value$, $addr$, deposit-a-list (list ($x0$, $x1$, $x2$, $x3$), $fp\text{-}addr$, $data\text{-}seg$))

THEOREM: lr-valp-car-p-temp-stk-p-run-subr-cons-helper
(lr-proper-heapp ($data\text{-}seg$)
$\wedge$ lr-valp (car, $car\text{-}addr$, $data\text{-}seg$)
$\wedge$ lr-valp (cdr, $cdr\text{-}addr$, $data\text{-}seg$)
$\wedge$ lr-proper-p-areasp ($data\text{-}seg$)
$\wedge$ (type ($ref\text{-}count$) = 'nat))
$\rightarrow$ lr-valp (cons (car, cdr),
fetch (identity (LR-FP-ADDR), $data\text{-}seg$),
deposit-a-list (list (identity (tag ('nat, LR-CONS-TAG)),
$ref\text{-}count$,
$car\text{-}addr$,
$cdr\text{-}addr$),
fetch (identity (LR-FP-ADDR), $data\text{-}seg$),
$data\text{-}seg$))

EVENT: Disable lr-valp-cons.

THEOREM: lr-valp-not-tag-true-not-listp
((¬ truep (*value*)) ∧ (fetch (*addr*, *data-seg*) = tag ('nat, LR-TRUE-TAG)))
→ (¬ lr-valp (*value*, *addr*, *data-seg*))

THEOREM: lr-valp-fetch-tag-not-true-lr-valp-listp
(proper-p-statep (lr->p (*l*))
∧ lr-proper-heapp (p-data-segment (*l*))
∧ (fetch (car (p-temp-stk (*l*)), p-data-segment (*l*))
≠ tag ('nat, LR-TRUE-TAG)))
→ (¬ lr-valp (**t**, car (p-temp-stk (*l*)), p-data-segment (*l*)))

THEOREM: lr-valp-car-p-temp-stk-p-run-subr
(lr-proper-heapp (p-data-segment (*l*))
∧ lr-check-result1 (reverse (*values*), p-temp-stk (*l*), p-data-segment (*l*))
∧ (length (*values*) = arity (*subr*))
∧ proper-p-statep (lr->p (*l*))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (p-run-subr (*subr*, p-set-pc (lr->p (*l*), *pc*))) = 'run)
∧ (p-psw (*l*) = 'run)
∧ (area-name (*pc*) = area-name (p-pc (*l*)))
∧ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (*l*))))))
= list ('call, *subr*)))
→ lr-valp (apply-subr (*subr*, *values*),
car (p-temp-stk (p-run-subr (*subr*, p-set-pc (lr->p (*l*), *pc*)))),
p-data-segment (p-run-subr (*subr*, p-set-pc (lr->p (*l*), *pc*))))

THEOREM: lr-programs-properp-not-definedp-subrp-runtime-support
(subrp (car (s-expr (*s*)))
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (¬ definedp (car (s-expr (*s*)), P-RUNTIME-SUPPORT-PROGRAMS))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*))))
→ (¬ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*))

THEOREM: lr-valp-apply-subr-lr-apply-subr
let *new-l* **be** lr-eval ('list, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)
in
(listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (car (s-expr (*s*)) ≠ 'quote)
∧ (car (s-expr (*s*)) ≠ S-TEMP-EVAL)
∧ (car (s-expr (*s*)) ≠ S-TEMP-TEST)
∧ (car (s-expr (*s*)) ≠ S-TEMP-FETCH)

$\wedge$ subrp (car (s-expr (s)))
$\wedge$ lr-proper-heapp (p-data-segment (*new-l*))
$\wedge$ lr-check-result1 (reverse (*values*),
p-temp-stk (*new-l*),
p-data-segment (*new-l*))
$\wedge$ (length (*values*) = length (cdr (lr-expr (s->lr1 (s, l, *table*)))))
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ s-good-statep (s, c)
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ (p-psw (lr-apply-subr (s->lr1 (s, l, *table*), *new-l*)) = `'run`)
$\wedge$ (*pos* = dv (s-pos (s), `1`)))
$\rightarrow$ lr-valp (apply-subr (car (s-expr (s)), *values*),
car (p-temp-stk (lr-apply-subr (s->lr1 (s, l, *table*), *new-l*))),
p-data-segment (lr-apply-subr (s->lr1 (s, l, *table*), *new-l*))) **endlet**

EVENT: Disable lr-programs-properp-not-definedp-subrp-runtime-support.

THEOREM: lr-check-result-lr-apply-subr
let *new-l* **be** lr-eval (`'list`,
lr-set-pos (s->lr1 (s, l, *table*), dv (s-pos (s), `1`)),
c),
pos **be** dv (s-pos (s), `1`)
in
(listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ `'if`)
$\wedge$ (car (s-expr (s)) $\neq$ `'quote`)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-TEST)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-FETCH)
$\wedge$ subrp (car (s-expr (s)))
$\wedge$ lr-check-result (`'list`,
s-ans (s-eval (`'list`, s-set-pos (s, *pos*), c)),
p-temp-stk (*new-l*),
p-data-segment (*new-l*),
p-temp-stk (l))
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ s-good-statep (s, c)
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ (p-psw (lr-apply-subr (s->lr1 (s, l, *table*), *new-l*)) = `'run`)

$\wedge$ (s-err-flag (s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c))
= 'run))
$\rightarrow$ lr-check-result (**t**,
apply-subr (car (s-expr (s)),
s-ans (s-eval ('list,
s-set-pos (s, *pos*),
c))),
p-temp-stk (lr-apply-subr (s->lr1 (s, l, *table*),
new-l)),
p-data-segment (lr-apply-subr (s->lr1 (s, l, *table*),
new-l)),
p-temp-stk (l)) **endlet**

THEOREM: s->lr1-lr-funcall-s-fun-call-state
(listp (s-expr (s))
$\wedge$ ($\neg$ subrp (car (s-expr (s))))
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ litatom (car (s-expr (s)))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ (p-psw (lr-funcall (s->lr1 (s, l, *table*), *lr-eval*)) = 'run)
$\wedge$ (s-progs (*s-eval*) = s-progs (s))
$\wedge$ (p-prog-segment (*lr-eval*) = p-prog-segment (s->lr1 (s, l, *table*))))
$\rightarrow$ (s->lr1 (s-fun-call-state (*s-eval*, car (s-expr (s))),
lr-funcall (s->lr1 (s, l, *table*), *lr-eval*),
table)
= lr-funcall (s->lr1 (s, l, *table*), *lr-eval*))

THEOREM: lr-params-lr-funcall
((p-psw (lr-funcall (*l1*, *l2*)) = 'run)
$\wedge$ (p-prog-segment (*l1*) = p-prog-segment (*l2*)))
$\rightarrow$ (lr-params (car (p-ctrl-stk (lr-funcall (*l1*, *l2*))), lr-funcall (*l1*, *l2*))
= pair-formal-vars-with-actuals (formal-vars (assoc (user-fname (car (lr-expr (*l1*))),
p-prog-segment (*l1*))),
p-temp-stk (*l2*)))

THEOREM: lr-temps-lr-funcall
((p-psw (lr-funcall (*l1*, *l2*)) = 'run)
$\wedge$ (p-prog-segment (*l1*) = p-prog-segment (*l2*)))
$\rightarrow$ (lr-temps (car (p-ctrl-stk (lr-funcall (*l1*, *l2*))), lr-funcall (*l1*, *l2*))
= pair-temps-with-initial-values (temp-var-dcls (assoc (user-fname (car (lr-expr (*l1*))),
p-prog-segment (*l1*)))))

THEOREM: listp-pairlist
listp (pairlist (x, y)) = listp (x)

THEOREM: car-reverse-last
car (reverse (*list*)) = car (last (*list*))

THEOREM: get-sub1-length-car-last
(listp (*list*) ∧ (*n* = (length (*list*) − 1)))
→ (get (*n*, *list*) = car (last (*list*)))

THEOREM: car-last-first-n-add1-get
car (last (first-n (1 + *n*, *list*))) = get (*n*, *list*)

THEOREM: length-butlast
length (butlast (*x*)) = (length (*x*) − 1)

DEFINITION:
induct-hint-1 (*x*, *y*, *z*)
= **if** listp (*x*)
then if listp (*y*)
then if listp (*z*)
then induct-hint-1 (cdr (*x*), butlast (*y*), butlast (*z*))
else t endif
else t endif
else t endif

THEOREM: lr-check-result1-append-2
(length (*values*) = length (*temp-stk1*))
→ (lr-check-result1 (*values*, append (*temp-stk1*, *temp-stk2*), *data-seg*)
= lr-check-result1 (*values*, *temp-stk1*, *data-seg*))

THEOREM: lr-check-result1-butlast
(lr-check-result1 (*values*, *temp-stk*, *data-seg*)
∧ (length (*temp-stk*) = length (*values*))
∧ listp (*temp-stk*)
∧ listp (*values*))
→ lr-check-result1 (butlast (*values*), butlast (*temp-stk*), *data-seg*)

THEOREM: reverse-butlast
listp (*x*) → (reverse (butlast (*x*)) = cdr (reverse (*x*)))

THEOREM: lr-s-similar-params-lr-valp-get
((*n* < length (*s-params*))
∧ (strip-cars (*s-params*) = strip-cars (*lr-params*))
∧ lr-s-similar-params (*s-params*, *lr-params*, *data-seg*))
→ lr-valp (cdr (get (*n*, *s-params*)), cdr (get (*n*, *lr-params*)), *data-seg*)

THEOREM: lr-s-similar-params-lr-funcall-helper-1
(lr-s-similar-params (pairlist (cdr (*formals*), cdr (reverse (*values*))),
pairlist (cdr (*formals*), cdr (reverse (*temp-stk*))),
data-seg)
∧ listp (*formals*)
∧ listp (*values*)
∧ listp (*temp-stk*)
∧ lr-check-result1 (*values*, *temp-stk*, *data-seg*)
∧ ((1 + length (cdr (*formals*))) = length (*temp-stk*))
∧ (length (*temp-stk*) = length (*values*)))
→ lr-valp (car (last (*values*)), car (last (*temp-stk*)), *data-seg*)

THEOREM: lr-s-similar-params-lr-funcall
(lr-check-result1 (*values*, *temp-stk*, *data-seg*)
∧ (length (*temp-stk*) = length (*values*))
∧ (length (*temp-stk*) = length (*formals*)))
→ lr-s-similar-params (pairlist (*formals*, reverse (*values*)),
pairlist (*formals*, reverse (*temp-stk*)),
data-seg)

THEOREM: append-first-n-restn
(length (*l*) ≮ *i*) → (append (first-n (*i*, *l*), restn (*i*, *l*)) = *l*)

THEOREM: lr-check-result1-first-n-temp-stk
(length (p-temp-stk (*l*)) ≮ length (*values*))
→ (lr-check-result1 (*values*, p-temp-stk (*l*), *data-seg*)
= lr-check-result1 (*values*,
first-n (length (*values*), p-temp-stk (*l*)),
data-seg))

THEOREM: lr-push-tstk-length
(p-psw (lr-push-tstk (*l*, *object*)) = 'run)
→ (length (p-temp-stk (lr-push-tstk (*l*, *object*)))
= (1 + length (p-temp-stk (*l*))))

THEOREM: length-add1-add1-cddr-fact
(length (*x*) = (1 + (1 + length (*y*)))) → (length (cddr (*x*)) = length (*y*))

THEOREM: length-p-temp-stk-p-run-subr-helper-1
(length (p-temp-stk (lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)))
= (1 + (1 + length (p-temp-stk (*l*)))))
→ (length (cddr (p-temp-stk (lr-eval ('list, lr-set-pos (*l*, *pos*), *c*))))
= length (p-temp-stk (*l*)))

THEOREM: length-p-temp-stk-p-run-subr

let *new-l* **be** lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)
in
(listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) ≠ 'if)
∧ subrp (car (lr-expr (*l*)))
∧ (length (p-temp-stk (*new-l*))
= (length (p-temp-stk (*l*)) + arity (car (lr-expr (*l*)))))
∧ proper-p-statep (lr->p (*l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*), lr-return-pc (*l*))))
= 'run)
∧ (p-psw (*new-l*) = 'run)
∧ (*pos* = dv (offset (p-pc (*l*)), 1)))
→ (length (p-temp-stk (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*),
lr-return-pc (*l*)))))
= (1 + length (p-temp-stk (*l*)))) **endlet**

THEOREM: length-p-temp-stk-lr-apply-subr
let *new-l* **be** lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)
in
(listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) ≠ 'if)
∧ subrp (car (lr-expr (*l*)))
∧ (length (p-temp-stk (*new-l*))
= (length (p-temp-stk (*l*)) + arity (car (lr-expr (*l*)))))
∧ proper-p-statep (lr->p (*l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (*new-l*) = 'run)
∧ (p-psw (lr-apply-subr (*l*, *new-l*)) = 'run)
∧ (*pos* = dv (offset (p-pc (*l*)), 1)))
→ (length (p-temp-stk (lr-apply-subr (*l*, *new-l*)))
= (1 + length (p-temp-stk (*l*)))) **endlet**

DEFINITION:
lr-proper-formalsp (*programs*)
= **if** listp (*programs*)
then ((logic-fname (name (car (*programs*))) = 'quote)
∨ (length (formal-vars (car (*programs*)))
= arity (logic-fname (name (car (*programs*))))))
∧ lr-proper-formalsp (cdr (*programs*))
else t endif

THEOREM: length-formal-vars-lr-proper-formalsp-arity
(definedp (*name*, *programs*)
 ∧ (logic-fname (*name*) ≠ 'quote)
 ∧ lr-proper-formalsp (*programs*))
→ (length (formal-vars (assoc (*name*, *programs*)))
 = arity (logic-fname (*name*)))

THEOREM: arity-formals-not-quote
(formals (*name*) ∧ (*name* ≠ 'quote))
→ (arity (*name*) = length (formals (*name*)))

THEOREM: lr-proper-formalsp-lr-compile-programs
s-programs-okp (*programs*)
→ lr-proper-formalsp (lr-compile-programs (*programs*, *table*))

EVENT: Disable arity-formals-not-quote.

EVENT: Disable lr-proper-formalsp.

THEOREM: lr-programs-properp-funcall-not-caar-prog-seg
(listp (lr-expr (*l*))
 ∧ (car (lr-expr (*l*)) ≠ 'if)
 ∧ (car (lr-expr (*l*)) ≠ 'quote)
 ∧ (¬ subrp (car (lr-expr (*l*))))
 ∧ litatom (car (lr-expr (*l*)))
 ∧ listp (p-prog-segment (*l*))
 ∧ (user-fname (car (lr-expr (*l*))) = caar (p-prog-segment (*l*)))
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*))))
→ (¬ lr-programs-properp (*l*, *table*))

THEOREM: length-p-temp-stk-lr-funcall
(listp (lr-expr (*l*))
 ∧ (car (lr-expr (*l*)) ≠ 'if)
 ∧ (car (lr-expr (*l*)) ≠ 'quote)
 ∧ (p-psw (*new-l*) = 'run)
 ∧ (¬ subrp (car (lr-expr (*l*))))
 ∧ litatom (car (lr-expr (*l*)))
 ∧ (length (p-temp-stk (lr-eval (**t**, lr-funcall (*l*, *new-l*), *c* − 1)))
 = (1 + length (p-temp-stk (lr-funcall (*l*, *new-l*)))))
 ∧ (length (p-temp-stk (*new-l*))
 = (length (p-temp-stk (*l*)) + arity (car (lr-expr (*l*)))))
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ lr-proper-formalsp (cdr (p-prog-segment (*l*)))

$\wedge$ (p-psw (lr-funcall (l, *new-l*)) = 'run))
$\rightarrow$ (length (p-temp-stk (lr-funcall (l, *new-l*))) = length (p-temp-stk (l)))

THEOREM: length-p-temp-stk-lr-eval
(proper-p-statep (lr->p (l))
$\wedge$ good-posp (*flag*, offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ lr-proper-formalsp (cdr (p-prog-segment (l)))
$\wedge$ (p-psw (lr-eval (*flag*, l, c)) = 'run))
$\rightarrow$ (length (p-temp-stk (lr-eval (*flag*, l, c)))
= **if** *flag* = 'list
then length (lr-expr-list (l)) + length (p-temp-stk (l))
else 1 + length (p-temp-stk (l)) **endif**)

THEOREM: length-p-temp-stk-lr-eval-flag-list
(proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ good-posp ('list, *pos*, s-body (s-prog (s)))
$\wedge$ s-good-statep (s, c)
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ (p-psw (lr-eval ('list, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c))
= 'run))
$\rightarrow$ (length (p-temp-stk (lr-eval ('list,
lr-set-pos (s->lr1 (s, l, *table*), *pos*),
c)))
= (length (s-expr-list (s-set-pos (s, *pos*)))
+ length (p-temp-stk (l))))

THEOREM: reverse-reverse-alt
reverse (reverse (l)) = plist (l)

THEOREM: pairlist-plist-1
pairlist (x, plist (y)) = pairlist (x, y)

THEOREM: s-good-statep-length-cdr-s-expr-funcall
(s-good-statep (s, c)
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ (litatom (car (s-expr (s))) $\vee$ subrp (car (s-expr (s)))))
$\rightarrow$ (length (cdr (s-expr (s))) = arity (car (s-expr (s))))

THEOREM: lr-s-similar-temps-make-temps-pair-temps
lr-s-similar-temps (make-temps-entries (*temp-list*),
pair-temps-with-initial-values (lr-make-temp-var-dcls (lr-make-temp-name-alist-1 (*initial*,

num-list,
temp-list
formals))

data-seg)

THEOREM: lr-s-similar-temps-lr-funcall
lr-s-similar-temps (make-temps-entries (s-temp-list (assoc (*name*, *progs*))),
pair-temps-with-initial-values (temp-var-dcls (assoc (*name*,
lr-compile-programs (*progs*,
table)))),
data-seg)

THEOREM: lr-eval-preserves-lr-s-similar-const-table
(proper-p-statep (lr->p (*l*))
∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table1*)
∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = 'run)
∧ lr-proper-heapp (p-data-segment (*l*))
∧ lr-s-similar-const-table (*table2*, p-data-segment (*l*)))
→ lr-s-similar-const-table (*table2*, p-data-segment (lr-eval (*flag*, *l*, *c*)))

THEOREM: lr-s-similar-statesp-lr-funcall
let *pos* **be** dv (s-pos (*s*), 1)
in
(listp (s-expr (*s*))
∧ (¬ subrp (car (s-expr (*s*))))
∧ (car (s-expr (*s*)) ≠ 'quote)
∧ (car (s-expr (*s*)) ≠ 'if)
∧ litatom (car (s-expr (*s*)))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ s-good-statep (*s*, *c*)
∧ (p-psw (lr-funcall (s->lr1 (*s*, *l*, *table*), *lr-eval*)) = 'run)
∧ (p-psw (*lr-eval*) = 'run)
∧ lr-check-result ('list,
values,
p-temp-stk (*lr-eval*),
p-data-segment (*lr-eval*),
p-temp-stk (*l*))
∧ lr-proper-heapp (p-data-segment (*l*))
∧ (s-err-flag (s-eval ('list, s-set-pos (*s*, *pos*), *c*)) = 'run)
∧ (*formals* = s-formals (assoc (user-fname (car (s-expr (*s*))),
s-progs (*s*))))
∧ (*lr-eval* = lr-eval ('list,

lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*),
c))
∧ (*values* = s-ans (s-eval (`'list`, s-set-pos (*s*, *pos*), *c*)))
∧ lr-s-similar-statesp (s-params (*s*),
s-temps (*s*),
s->lr1 (*s*, *l*, *table*),
table))
→ lr-s-similar-statesp (pairlist (*formals*, *values*),
make-temps-entries (s-temp-list (assoc (user-fname (car (s-expr (*s*))),
s-progs (*s*)))),
lr-funcall (s->lr1 (*s*, *l*, *table*), *lr-eval*),
table) **endlet**

THEOREM: lr-params-lr-set-expr-lr-pop-cstk
((area-name (p-pc (*l*)) = area-name (p-pc (*new-l*)))
∧ (p-prog-segment (*l*) = p-prog-segment (*new-l*)))
→ (lr-params (*frame*,
lr-set-expr (lr-pop-cstk (lr-eval (**t**, lr-funcall (*l*, *new-l*), *c*)),
l,
pos))
= lr-params (*frame*, *l*))

THEOREM: lr-temps-lr-set-expr-lr-pop-cstk
((area-name (p-pc (*l*)) = area-name (p-pc (*new-l*)))
∧ (p-prog-segment (*l*) = p-prog-segment (*new-l*)))
→ (lr-temps (*frame*,
lr-set-expr (lr-pop-cstk (lr-eval (**t**, lr-funcall (*l*, *new-l*), *c*)),
l,
pos))
= lr-temps (*frame*, *l*))

THEOREM: lr-eval-preserves-lr-s-similar-params
(proper-p-statep (lr->p (*l*))
∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = `'run`)
∧ lr-proper-heapp (p-data-segment (*l*))
∧ lr-s-similar-params (*s-params*, *lr-params*, p-data-segment (*l*)))
→ lr-s-similar-params (*s-params*,
lr-params,
p-data-segment (lr-eval (*flag*, *l*, *c*)))

THEOREM: lr-eval-preserves-lr-s-similar-temps
(proper-p-statep (lr->p (*l*))
∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))

$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval (*flag*, l, c)) = 'run)
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ lr-s-similar-temps (*s-temps*, *lr-temps*, p-data-segment (l)))
$\rightarrow$ lr-s-similar-temps (*s-temps*, *lr-temps*, p-data-segment (lr-eval (*flag*, l, c)))

THEOREM: lr-s-similar-statesp-lr-set-expr-lr-pop-cstk
let *funcall* **be** lr-funcall (s->lr1 (s, l, *table*),
lr-eval ('list,
lr-set-pos (s->lr1 (s, l, *table*), *pos*),
c)),
lr-eval **be** lr-eval ('list, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c)
in
(lr-s-similar-statesp (s-params (s), s-temps (*s-eval*), *lr-eval*, *table*)
$\wedge$ listp (s-expr (s))
$\wedge$ ($\neg$ subrp (car (s-expr (s))))
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ litatom (car (s-expr (s)))
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ lr-proper-heapp (p-data-segment (*lr-eval*))
$\wedge$ (*pos* = dv (s-pos (s), 1))
$\wedge$ (p-psw (lr-eval (**t**, *funcall*, $c - 1$)) = 'run)
$\wedge$ (p-psw (*lr-eval*) = 'run))
$\rightarrow$ lr-s-similar-statesp (s-params (s),
s-temps (*s-eval*),
lr-set-expr (lr-pop-cstk (lr-eval (**t**,
funcall,
$c - 1$)),
s->lr1 (s, l, *table*),
s-pos (s)),
table) **endlet**

THEOREM: popn-restn
(length (*list*) $\not<$ n) $\rightarrow$ (popn (n, *list*) = restn (n, *list*))

THEOREM: lr-check-result-lr-funcall
let *new-l* **be** lr-eval ('list,
lr-set-pos (s->lr1 (s, l, *table*), dv (s-pos (s), 1)),
c),
pos **be** dv (s-pos (s), 1),
s-eval **be** s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)
in

```
(listp (s-expr (s))
 ∧   (car (s-expr (s)) ≠ 'if)
 ∧   (car (s-expr (s)) ≠ 'quote)
 ∧   (¬ subrp (car (s-expr (s))))
 ∧   litatom (car (s-expr (s)))
 ∧   lr-check-result ('list,
                      s-ans (s-eval),
                      p-temp-stk (new-l),
                      p-data-segment (new-l),
                      p-temp-stk (l))
 ∧   proper-p-statep (lr->p (s->lr1 (s, l, table)))
 ∧   good-posp1 (s-pos (s), s-body (s-prog (s)))
 ∧   lr-programs-properp (s->lr1 (s, l, table), table)
 ∧   s-good-statep (s, c)
 ∧   (p-psw (new-l) = 'run)
 ∧   (p-psw (lr-funcall (s->lr1 (s, l, table), new-l)) = 'run)
 ∧   (s-err-flag (s-eval) = 'run)
 ∧   lr-check-result (t,
                      s-ans (s-eval (t,
                                     s-fun-call-state (s-eval,
                                                       car (s-expr (s))),
                                     c − 1)),
                      p-temp-stk (lr-eval (t,
                                           lr-funcall (s->lr1 (s,
                                                               l,
                                                               table),
                                                       new-l),
                                           c − 1)),
                      p-data-segment (lr-eval (t,
                                               lr-funcall (s->lr1 (s,
                                                                   l,
                                                                   table),
                                                           new-l),
                                               c − 1)),
                      p-temp-stk (lr-funcall (s->lr1 (s, l, table), new-l))))
 →   lr-check-result (t,
                      s-ans (s-eval (t,
                                     s-fun-call-state (s-eval,
                                                       car (s-expr (s))),
                                     c − 1)),
                      p-temp-stk (lr-eval (t,
                                           lr-funcall (s->lr1 (s, l, table),
                                                       new-l),
                                           c − 1)),
```

p-data-segment (lr-eval (**t**,
lr-funcall (s->lr1 (*s*,
l,
table),
new-l),
c − 1)),
p-temp-stk (*l*)) **endlet**

EVENT: Disable popn-restn.

THEOREM: lr-eval-s->lr1-flag-list-opener-1
(good-posp (`'list`, s-pos (*s*), s-body (s-prog (*s*)))
∧ listp (s-expr-list (*s*))
∧ listp (s-pos (*s*))
∧ (s-err-flag (*s*) = `'run`))
→ (lr-eval (`'list`, s->lr1 (*s*, *l*, *table*), *c*)
= lr-eval (`'list`,
lr-set-expr (lr-eval (**t**, s->lr1 (*s*, *l*, *table*), *c*),
s->lr1 (*s*, *l*, *table*),
nx (s-pos (*s*))),
c))

THEOREM: lr-eval-s->lr1-flag-list-opener-2
(good-posp (`'list`, s-pos (*s*), s-body (s-prog (*s*)))
∧ (¬ listp (s-expr-list (*s*)))
∧ listp (s-pos (*s*))
∧ (s-err-flag (*s*) = `'run`))
→ (lr-eval (`'list`, s->lr1 (*s*, *l*, *table*), *c*) = s->lr1 (*s*, *l*, *table*))

THEOREM: lr-check-result-lr-proper-heapp
lr-check-result (*flag*, *value*, *temp-stk*, *data-seg*, *orig-temp-stk*)
→ lr-proper-heapp (*data-seg*)

THEOREM: lr-programs-properp-lr-set-error
lr-programs-properp (lr-set-error (*l*, *error*), *table*)
= lr-programs-properp (*l*, *table*)

THEOREM: p-psw-lr-pop-tstk-lr-eval-flag-t
(proper-p-statep (lr->p (*l*))
∧ good-posp1 (*pos*, program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (lr-if-ok (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*))) = `'run`))
→ (p-psw (lr-pop-tstk (lr-if-ok (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)))))
= `'run`)

THEOREM: lr-eval-leaves-listp-p-temp-stk-lr-set-pos
(proper-p-statep (lr->p (*l*))
 ∧ good-posp1 (*pos*, program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)) = 'run))
→ listp (p-temp-stk (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)))

THEOREM: p-psw-run-lr-if-ok-p-psw-run
(p-psw (lr-if-ok (*l*)) = 'run) → (p-psw (*l*) = 'run)

THEOREM: lr-s-similar-statesp-lr-if-ok
lr-s-similar-statesp (*s-params*, *s-temps*, lr-if-ok (*l*), *table*)
= lr-s-similar-statesp (*s-params*, *s-temps*, *l*, *table*)

THEOREM: lr-eval-s-eval-equivalence
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
 ∧ s-good-statep (*s*, *c*)
 ∧ (p-psw (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)) = 'run))
→ (lr-s-similar-statesp (s-params (*s*),
 s-temps (s-eval (*flag*, *s*, *c*)),
 lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*),
 table)
 ∧ lr-check-result (**if** *flag* = 'list **then** 'list
 else t endif,
 s-ans (s-eval (*flag*, *s*, *c*)),
 p-temp-stk (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)),
 p-data-segment (lr-eval (*flag*,
 s->lr1 (*s*, *l*, *table*),
 c)),
 p-temp-stk (*l*))
 ∧ (s-err-flag (s-eval (*flag*, *s*, *c*)) = 'run))

EVENT: Disable p-psw-run-lr-if-ok-p-psw-run.

```
; -------------------------------------------------------------
; was lr-eval5.events
; -------------------------------------------------------------


;; The following define functions for each SUBR that tell how many
```

```
;; resources are used.  In the computations of the maximum control
;; stack size we break out the parts needed for formals and
;; temporaries and building a new control-stack frame.  For example in
;; CONS we have (plus 2 0 1 ...), the 2 is for building a new frame,
;; the 0 is for the formals (CONS leaves its args on the temp stack)
;; and 1 for temporaries.
```

DEFINITION:
s-apply-car-r (s) = list (1, 2 + 1 + 0 + 0, 0, 0)

DEFINITION:
s-apply-cdr-r (s) = list (1, 2 + 1 + 0 + 0, 0, 0)

```
;; CONS takes two implicit args
```

DEFINITION:
s-apply-cons-r (s) = list (2, 2 + 0 + 1 + 0, 0, 1)

DEFINITION:
s-apply-false-r (s) = list (1, 2 + 0 + 0 + 0, 0, 0)

```
;; FALSEP takes one implicit arg on stack.
```

DEFINITION:
s-apply-falsep-r (s) = list (1, 2 + 0 + 0 + 0, 0, 0)

```
;; LISTP takes an implicit arg
```

DEFINITION:
s-apply-listp-r (s) = list (1, 2 + 0 + 0 + 0, 0, 0)

```
;; NLISTP takes an implicit arg
```

DEFINITION:
s-apply-nlistp-r (s) = list (1, 2 + 0 + 0 + 0, 0, 0)

DEFINITION:
s-apply-true-r (s) = list (1, 2 + 0 + 0 + 0, 0, 0)

DEFINITION:
s-apply-truep-r (s) = list (1, 2 + 0 + 0 + 0, 0, 0)

DEFINITION:
s-apply-subr-r $(subr, s)$
= **case on** *subr*:
case = *car*
then s-apply-car-r (s)

case = *cdr*
then s-apply-cdr-r (*s*)
case = *cons*
then s-apply-cons-r (*s*)
case = *false*
then s-apply-false-r (*s*)
case = *falsep*
then s-apply-falsep-r (*s*)
case = *listp*
then s-apply-listp-r (*s*)
case = *nlistp*
then s-apply-nlistp-r (*s*)
case = *true*
then s-apply-true-r (*s*)
case = *truep*
then s-apply-truep-r (*s*)
otherwise list (0, 0, 0, 0) **endcase**

DEFINITION:
max-r (*list1*, *list2*)
= list (max (car (*list1*), car (*list2*)),
max (cadr (*list1*), cadr (*list2*)),
max (caddr (*list1*), caddr (*list2*)),
cadddr (*list1*) + cadddr (*list2*))

EVENT: Disable max-r.

DEFINITION:
s-add-temp-r (*list*, *n*)
= list (*n* + car (*list*), cadr (*list*), caddr (*list*), cadddr (*list*))

```
;; S-EVAL-R is somewhat similar to S-EVAL.  It returns a list of four
;; numbers representing.  the maximum temp stack size, maximum ctrl stack
;; size, maximum word size and number of free heap nodes respectively needed
;; to execute the compilation of the S-STATE s in Piton without getting an
;; error.
```

DEFINITION:
s-eval-r (*flag*, *s*, *c*)
= **if** s-err-flag (*s*) ≠ 'run **then** list (0, 0, 0, 0)
elseif *flag* = 'list
then if s-pos (*s*) ≃ **nil** **then** list (0, 0, 0, 0)
elseif listp (s-expr-list (*s*))
then max-r (s-eval-r (**t**, *s*, *c*),

```
                    s-add-temp-r (s-eval-r ('list,
                                            s-set-expr (s-eval (t, s, c),
                                                        s,
                                                        nx (s-pos (s))),
                                            c),
                                  1))
      else list (0, 0, 0, 0) endif
elseif c ≃ 0 then list (0, 0, 0, 0)
elseif litatom (s-expr (s)) then list (1, 0, 0, 0)
elseif s-expr (s) ≃ nil then list (0, 0, 0, 0)
elseif car (s-expr (s)) = 'if
then let test be s-eval (t, s-set-pos (s, dv (s-pos (s), 1)), c)
      in
      if s-err-flag (test) = 'run
      then if s-ans (test)
           then max-r (s-add-temp-r (s-eval-r (t,
                                               s-set-pos (s,
                                                          dv (s-pos (s),
                                                              1)),
                                               c),
                                     1),
                       s-eval-r (t,
                                 s-set-expr (test,
                                             s,
                                             dv (s-pos (s), 2)),
                                 c))
           else max-r (s-add-temp-r (s-eval-r (t,
                                               s-set-pos (s,
                                                          dv (s-pos (s),
                                                              1)),
                                               c),
                                     1),
                       s-eval-r (t,
                                 s-set-expr (test,
                                             s,
                                             dv (s-pos (s), 3)),
                                 c)) endif
      else s-eval-r (t, s-set-pos (s, dv (s-pos (s), 1)), c) endif endlet
elseif car (s-expr (s)) = S-TEMP-EVAL
then s-eval-r (t, s-set-pos (s, dv (s-pos (s), 1)), c)
elseif car (s-expr (s)) = S-TEMP-TEST
then if s-temp-setp (cadr (s-expr (s)), s-temps (s)) then list (2, 0, 0, 0)
      else max-r (list (2, 0, 0, 0),
                  s-eval-r (t, s-set-pos (s, dv (s-pos (s), 1)), c)) endif
```

```
elseif car (s-expr (s)) = S-TEMP-FETCH then list (1, 0, 0, 0)
elseif car (s-expr (s)) = 'quote then list (1, 0, 0, 0)
elseif s-err-flag (s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c))
       ≠  'run
then s-eval-r ('list, s-set-pos (s, dv (s-pos (s), 1)), c)
elseif subrp (car (s-expr (s)))
then max-r (s-eval-r ('list, s-set-pos (s, dv (s-pos (s), 1)), c),
                s-add-temp-r (s-apply-subr-r (car (s-expr (s)),
                                              s-eval ('list,
                                                      s-set-pos (s,
                                                                 dv (s-pos (s),
                                                                     1)),
                                                      c)),
                                   arity (car (s-expr (s)))))
elseif litatom (car (s-expr (s)))
then let arg-s be s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)
     in
     let fstate be s-fun-call-state (arg-s, car (s-expr (s)))
     in
     let arg-r be s-eval-r (t, fstate, c − 1)
     in
     max-r (s-eval-r ('list,
                      s-set-pos (s, dv (s-pos (s), 1)),
                      c),
            list (car (arg-r),
                  2
                  +  length (s-params (fstate))
                  +  length (s-temps (fstate))
                  +  cadr (arg-r),
                  caddr (arg-r),
                  cadddr (arg-r))) endlet endlet endlet
else list (0, 0, 0, 0) endif
```

DEFINITION: s-eval-temp-r (*flag*, *s*, *c*) = car (s-eval-r (*flag*, *s*, *c*))

DEFINITION: s-eval-ctrl-r (*flag*, *s*, *c*) = cadr (s-eval-r (*flag*, *s*, *c*))

DEFINITION: s-eval-ws-r (*flag*, *s*, *c*) = caddr (s-eval-r (*flag*, *s*, *c*))

DEFINITION:
s-eval-heap-r (*flag*, *s*, *c*) = cadddr (s-eval-r (*flag*, *s*, *c*))

DEFINITION:
S-MAX-SUBR-REQS
= max (log (2, LR-CONS-TAG),

max (log (2, LR-TRUE-TAG),
max (log (2, LR-CDR-OFFSET), log (2, LR-CAR-OFFSET))))

EVENT: Disable s-max-subr-reqs.

THEOREM: numberp-car-cadr-caddr-cadddr-s-apply-subr-r
(car (s-apply-subr-r (*subr*, *s*)) ∈ **N**)
∧ (cadr (s-apply-subr-r (*subr*, *s*)) ∈ **N**)
∧ (caddr (s-apply-subr-r (*subr*, *s*)) ∈ **N**)
∧ (cadddr (s-apply-subr-r (*subr*, *s*)) ∈ **N**)

EVENT: Disable s-apply-subr-r.

THEOREM: numberp-max-r
(car (max-r (*list1*, *list2*)) ∈ **N**)
∧ (cadr (max-r (*list1*, *list2*)) ∈ **N**)
∧ (caddr (max-r (*list1*, *list2*)) ∈ **N**)
∧ (cadddr (max-r (*list1*, *list2*)) ∈ **N**)

THEOREM: numberp-s-eval-temp-ctrl-ws-heap-r
(s-eval-temp-r (*flag*, *s*, *c*) ∈ **N**)
∧ (s-eval-ctrl-r (*flag*, *s*, *c*) ∈ **N**)
∧ (s-eval-ws-r (*flag*, *s*, *c*) ∈ **N**)
∧ (s-eval-heap-r (*flag*, *s*, *c*) ∈ **N**)

EVENT: Disable s-eval-temp-r.

EVENT: Disable s-eval-ctrl-r.

EVENT: Disable s-eval-ws-r.

EVENT: Disable s-eval-heap-r.

DEFINITION:
lr-count-free-nodes (*addr*, *node-list*, *data-seg*)
= **if** *addr* ∈ *node-list*
then 1 + lr-count-free-nodes (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET),
data-seg),
delete (*addr*, *node-list*),
data-seg)
else 0 **endif**

DEFINITION:
lr-check-resourcesp (*flag*, *s*, *l*, *c*)
= ((p-max-temp-stk-size (*l*)
$\not<$ (length (p-temp-stk (*l*)) + s-eval-temp-r (*flag*, *s*, *c*)))
$\wedge$ (p-max-ctrl-stk-size (*l*)
$\not<$ (p-ctrl-stk-size (p-ctrl-stk (*l*))
+ s-eval-ctrl-r (*flag*, *s*, *c*)))
$\wedge$ (p-word-size (*l*) $\not<$ s-eval-ws-r (*flag*, *s*, *c*))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (*l*)),
lr-free-list-nodes (lr-max-node (p-data-segment (*l*)),
p-data-segment (*l*)),
p-data-segment (*l*))
$\not<$ s-eval-heap-r (*flag*, *s*, *c*)))

EVENT: Disable lr-check-resourcesp.

THEOREM: not-lessp-max-r-car
(car (max-r (*list1*, *list2*)) $\not<$ car (*list1*))
$\wedge$ (car (max-r (*list1*, *list2*)) $\not<$ car (*list2*))

THEOREM: not-lessp-max-r-cadr
(cadr (max-r (*list1*, *list2*)) $\not<$ cadr (*list1*))
$\wedge$ (cadr (max-r (*list1*, *list2*)) $\not<$ cadr (*list2*))

THEOREM: not-lessp-max-r-caddr
(caddr (max-r (*list1*, *list2*)) $\not<$ caddr (*list1*))
$\wedge$ (caddr (max-r (*list1*, *list2*)) $\not<$ caddr (*list2*))

THEOREM: not-lessp-max-r-cadddr
(cadddr (max-r (*list1*, *list2*)) $\not<$ cadddr (*list1*))
$\wedge$ (cadddr (max-r (*list1*, *list2*)) $\not<$ cadddr (*list2*))

THEOREM: lr-check-resourcesp-listp-s-expr-list
((s-err-flag (*s*) = 'run)
$\wedge$ listp (s-pos (*s*))
$\wedge$ listp (s-expr-list (*s*))
$\wedge$ good-posp ('list, s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ lr-check-resourcesp ('list, *s*, *l*, *c*))
$\rightarrow$ lr-check-resourcesp (**t**, *s*, *l*, *c*)

THEOREM: lr-eval-preserves-lr-proper-heapp
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
$\wedge$ lr-proper-heapp (p-data-segment (*l*))
$\wedge$ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)

$\wedge$ lr-s-similar-statesp (s-params (s), s-temps (s), s->lr1 (s, l, *table*), *table*)
$\wedge$ s-good-statep (s, c)
$\wedge$ (p-psw (lr-eval (*flag*, s->lr1 (s, l, *table*), c)) = 'run))
$\rightarrow$ lr-proper-heapp (p-data-segment (lr-eval (*flag*, s->lr1 (s, l, *table*), c)))

THEOREM: lr-eval-preserves-lr-s-similar-statesp
(proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ good-posp (*flag*, s-pos (s), s-body (s-prog (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ lr-s-similar-statesp (s-params (s), s-temps (s), s->lr1 (s, l, *table*), *table*)
$\wedge$ s-good-statep (s, c)
$\wedge$ (p-psw (lr-eval (*flag*, s->lr1 (s, l, *table*), c)) = 'run))
$\rightarrow$ lr-s-similar-statesp (s-params (s),
s-temps (s-eval (*flag*, s, c)),
lr-eval (*flag*, s->lr1 (s, l, *table*), c),
table)

THEOREM: s-eval-flag-run-flag-t-subsetp-s-collect-all-temps
(s-good-statep (s, c)
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ s-all-temps-setp (*flag*, s-expr (s), temp-alist-to-set (s-temps (s)))
$\wedge$ s-all-progs-temps-setp (s-progs (s))
$\wedge$ (s-err-flag (s-eval (*flag*, s, c)) = 'run)
$\wedge$ s-check-temps-setp (s-temps (s))
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ subsetp (s-collect-all-temps (*flag*, s-expr (s)),
temp-alist-to-set (s-temps (s-eval (*flag*, s, c))))

THEOREM: s-eval-flag-run-flag-t-s-check-temps-setp
(s-good-statep (s, c)
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ s-all-temps-setp (*flag*, s-expr (s), temp-alist-to-set (s-temps (s)))
$\wedge$ s-all-progs-temps-setp (s-progs (s))
$\wedge$ (s-err-flag (s-eval (*flag*, s, c)) = 'run)
$\wedge$ s-check-temps-setp (s-temps (s))
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ s-check-temps-setp (s-temps (s-eval (*flag*, s, c)))

THEOREM: lr-eval-preserves-length-bindings-car-p-ctrl-stk
(proper-p-statep (lr->p (l))
$\wedge$ good-posp (*flag*, offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval (*flag*, l, c)) = 'run))
$\rightarrow$ (length (bindings (car (p-ctrl-stk (lr-eval (*flag*, l, c))))))
= length (bindings (car (p-ctrl-stk (l)))))

THEOREM: lr-eval-s->lr1-preserves-p-ctrl-stk-size
(proper-p-statep (lr->p (*l*))
 ∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (p-psw (lr-eval (*flag*, *l*, *c*)) = 'run))
→ (p-ctrl-stk-size (p-ctrl-stk (lr-eval (*flag*, *l*, *c*)))
 = p-ctrl-stk-size (p-ctrl-stk (*l*)))

THEOREM: length-p-temp-stk-lr-eval-flag-not-list
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
 ∧ s-good-statep (*s*, *c*)
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ (p-psw (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)) = 'run)
 ∧ (*flag* ≠ 'list))
→ (length (p-temp-stk (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)))
 = (1 + length (p-temp-stk (*l*))))

THEOREM: lr-eval-preserves-lr-proper-heapp-lr-set-pos
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ good-posp (*flag*, *pos*, s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
 ∧ s-good-statep (*s*, *c*)
 ∧ (p-psw (lr-eval (*flag*, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)) = 'run))
→ lr-proper-heapp (p-data-segment (lr-eval (*flag*,
 lr-set-pos (s->lr1 (*s*, *l*, *table*),
 pos),
 c)))

THEOREM: lr-eval-preserves-lr-s-similar-statesp-lr-set-pos
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ good-posp (*flag*, *pos*, s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
 ∧ s-good-statep (*s*, *c*)
 ∧ (p-psw (lr-eval (*flag*, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)) = 'run))
→ lr-s-similar-statesp (s-params (*s*),
 s-temps (s-eval (*flag*, s-set-pos (*s*, *pos*), *c*)),
 lr-eval (*flag*, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*),
 table)

THEOREM: lr-eval-s-eval-flag-t-s-ans-f-lr-set-pos

(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ good-posp1 (*pos*, s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
 ∧ s-good-statep (*s*, *c*)
 ∧ (p-psw (lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)) = 'run)
 ∧ (¬ s-ans (s-eval (**t**, s-set-pos (*s*, *pos*), *c*))))
→ (car (p-temp-stk (lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)))
 = LR-F-ADDR)

THEOREM: lr-eval-s-eval-flag-t-s-ans-non-f-lr-set-pos
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ good-posp1 (*pos*, s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
 ∧ s-good-statep (*s*, *c*)
 ∧ (p-psw (lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)) = 'run)
 ∧ s-ans (s-eval (**t**, s-set-pos (*s*, *pos*), *c*)))
→ (car (p-temp-stk (lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)))
 ≠ identity (LR-F-ADDR))

THEOREM: subsetp-not-member-both
((*addr* ∉ *set2*) ∧ subsetp (*set1*, *set2*)) → (*addr* ∉ *set1*)

THEOREM: lr-count-free-nodes-deposit-free-ptr
(adpp ('(free-ptr . 0), *data-seg*) ∧ lr-node-listp (*node-list*, *data-seg*))
→ (lr-count-free-nodes (*addr*,
 node-list,
 deposit (*anything*, identity (LR-FP-ADDR), *data-seg*))
 = lr-count-free-nodes (*addr*, *node-list*, *data-seg*))

THEOREM: lr-count-free-nodes-deposit-non-ref-count
(lr-nodep (*addr2*, *data-seg*)
 ∧ (*offset* ≠ LR-REF-COUNT-OFFSET)
 ∧ (*offset* ∈ **N**)
 ∧ (*offset* < LR-NODE-SIZE)
 ∧ lr-node-listp (*node-list*, *data-seg*))
→ (lr-count-free-nodes (*addr1*,
 node-list,
 deposit (*anything*, add-addr (*addr2*, *offset*), *data-seg*))
 = lr-count-free-nodes (*addr1*, *node-list*, *data-seg*))

THEOREM: lr-count-free-nodes-deposit-lr-nodep

(lr-nodep (*addr2*, *data-seg*) ∧ lr-node-listp (*node-list*, *data-seg*))
→ (lr-count-free-nodes (*addr1*, *node-list*, deposit (*anything*, *addr2*, *data-seg*))
= lr-count-free-nodes (*addr1*, *node-list*, *data-seg*))

THEOREM: lr-count-free-nodes-delete-deposit
((*addr1* ∉ *node-list*)
∧ lr-nodep (*addr1*, *data-seg*)
∧ lr-node-listp (*node-list*, *data-seg*)
∧ (type (*ref-count*) = 'nat)
∧ (untag (*ref-count*) ∈ **N**))
→ (lr-count-free-nodes (*addr2*,
node-list,
deposit (*ref-count*,
add-addr (*addr1*,
identity (LR-REF-COUNT-OFFSET)),
data-seg))
= lr-count-free-nodes (*addr2*, *node-list*, *data-seg*))

THEOREM: lr-count-free-nodes-max-addr-lr-free-list-nodes
lr-count-free-nodes (*max-addr*,
lr-free-list-nodes (*max-addr*, *data-seg1*),
data-seg2)
= 0

THEOREM: lr-count-lr-free-list-nodes-p-run-cons
let *dds* **be** deposit-a-list (list (identity (tag ('nat, 5)),
ref-count,
any1,
any2),
fetch (identity (LR-FP-ADDR), *data-seg*),
data-seg)
in
(lr-proper-heapp (*data-seg*)
∧ (*max-addr* = lr-max-node (*data-seg*))
∧ (type (*ref-count*) = 'nat)
∧ (untag (*ref-count*) ∈ **N**)
∧ (fetch (identity (LR-FP-ADDR), *data-seg*) ≠ *max-addr*))
→ ((1 + lr-count-free-nodes (fetch (add-addr (fetch (identity (LR-FP-ADDR),
data-seg),
identity (LR-REF-COUNT-OFFSET)),
data-seg),
lr-free-list-nodes (*max-addr*, *dds*),
dds))
= lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
data-seg),

lr-free-list-nodes (*max-addr*,
data-seg),
data-seg)) **endlet**

THEOREM: not-p-max-node-fetch-fp-addr-not-errorp-p-run-cons
((p-psw (p (p-set-pc (lr->p (*new-l*), *pc*), p-cons-clock (p-set-pc (lr->p (*new-l*), *pc*))))
= 'run)
∧ proper-p-statep (lr->p (*new-l*))
∧ proper-p-statep (p-set-pc (lr->p (*new-l*), *pc*))
∧ (*max-node* = lr-max-node (p-data-segment (*new-l*)))
∧ (p-psw (*new-l*) = 'run)
∧ lr-programs-properp (*new-l*, *table*)
∧ lr-proper-heapp (p-data-segment (*new-l*))
∧ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (*new-l*))))))
= '(call cons)))
→ (fetch (identity (LR-FP-ADDR), p-data-segment (*new-l*)) ≠ *max-node*)

THEOREM: get-comp-body-lr-compile-programs
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (car (s-expr (*s*)) ≠ S-TEMP-EVAL)
∧ (car (s-expr (*s*)) ≠ S-TEMP-TEST)
∧ (car (s-expr (*s*)) ≠ S-TEMP-FETCH)
∧ (car (s-expr (*s*)) ≠ 'quote)
∧ listp (s-expr (*s*)))
→ (get (offset (lr-return-pc (s->lr1 (*s*, *l*, *table*))),
program-body (assoc (s-pname (*s*),
comp-programs (lr-compile-programs (s-progs (*s*),
table)))))
= list ('dl,
lr-make-label (offset (lr-return-pc (s->lr1 (*s*, *l*, *table*)))),
nil,
if definedp (car (s-expr (*s*)), P-RUNTIME-SUPPORT-PROGRAMS)
then list ('call, car (s-expr (*s*)))
else list ('call, user-fname (car (s-expr (*s*)))) **endif**))

THEOREM: lr-count-lr-free-list-nodes-p-run-subr
let *p* **be** p-set-pc (lr->p (lr-eval ('list,
lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*),
c)),
lr-return-pc (s->lr1 (*s*, *l*, *table*))),
new-l **be** lr-eval ('list, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)

in
(listp (s-expr (*s*))
 ∧ (car (s-expr (*s*)) ≠ 'if)
 ∧ subrp (car (s-expr (*s*)))
 ∧ proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ s-good-statep (*s*, *c*)
 ∧ (p-psw (p-run-subr (car (s-expr (*s*)), *p*)) = 'run)
 ∧ (p-psw (*new-l*) = 'run)
 ∧ (*pos* = dv (s-pos (*s*), 1))
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ lr-s-similar-statesp (s-params (*s*),
 s-temps (*s*),
 s->lr1 (*s*, *l*, *table*),
 table))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
 p-data-segment (*new-l*)),
 lr-free-list-nodes (lr-max-node (p-data-segment (*l*)),
 p-data-segment (*new-l*)),
 p-data-segment (*new-l*))
 = (lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
 p-data-segment (p-run-subr (car (s-expr (*s*)),
 p))),
 lr-free-list-nodes (lr-max-node (p-data-segment (*l*)),
 p-data-segment (p-run-subr (car (s-expr (*s*)),
 p))),
 p-data-segment (p-run-subr (car (s-expr (*s*)),
 p)))
 + cadddr (s-apply-subr-r (car (s-expr (*s*)),
 s-eval ('list,
 s-set-pos (*s*, *pos*),
 c))))) **endlet**

THEOREM: lr-count-lr-free-list-nodes-lr-apply-subr
let *new-l* **be** lr-eval ('list, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)
in
(listp (s-expr (*s*))
 ∧ (car (s-expr (*s*)) ≠ 'if)
 ∧ subrp (car (s-expr (*s*)))
 ∧ proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ (p-psw (*new-l*) = 'run)

$$
\begin{array}{ll}
\wedge & (\text{p-psw}\,(\text{lr-apply-subr}\,(\text{s->lr1}\,(s,\ l,\ \textit{table}),\ \textit{new-l})) = \texttt{'run}) \\
\wedge & \text{s-good-statep}\,(s,\ c) \\
\wedge & \text{lr-proper-heapp}\,(\text{p-data-segment}\,(l)) \\
\wedge & \text{lr-s-similar-statesp}\,(\text{s-params}\,(s), \\
 & \qquad\qquad\qquad\qquad \text{s-temps}\,(s), \\
 & \qquad\qquad\qquad\qquad \text{s->lr1}\,(s,\ l,\ \textit{table}), \\
 & \qquad\qquad\qquad\qquad \textit{table}) \\
\wedge & (\textit{pos} = \text{dv}\,(\text{s-pos}\,(s),\ 1)) \\
\wedge & (\textit{max-addr} = \text{lr-max-node}\,(\text{p-data-segment}\,(l))) \\
\wedge & (\textit{s-eval-size} = \text{s-eval-heap-r}\,(\texttt{'list},\ \text{s-set-pos}\,(s,\ \textit{pos}),\ c)) \\
\wedge & ((\text{lr-count-free-nodes}\,(\text{fetch}\,(\textsc{lr-fp-addr},\ \text{p-data-segment}\,(\textit{new-l})), \\
 & \qquad\qquad\qquad\qquad \text{lr-free-list-nodes}\,(\textit{max-addr}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(\textit{new-l})), \\
 & \qquad\qquad\qquad\qquad \text{p-data-segment}\,(\textit{new-l})) \\
 & \quad + \quad \textit{s-eval-size}) \\
 & \quad = \quad \text{lr-count-free-nodes}\,(\text{fetch}\,(\textsc{lr-fp-addr}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(l)), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{lr-free-list-nodes}\,(\textit{max-addr}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(l)), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(l)))) \\
\rightarrow & ((\textit{s-eval-size} \\
 & \quad + \quad \text{cadddr}\,(\text{s-apply-subr-r}\,(\text{car}\,(\text{s-expr}\,(s)), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{s-eval}\,(\texttt{'list}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{s-set-pos}\,(s,\ \textit{pos}), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c))) \\
 & \quad + \quad \text{lr-count-free-nodes}\,(\text{fetch}\,(\text{identity}\,(\textsc{lr-fp-addr}), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(\text{lr-apply-subr}\,(\text{s->lr1}\,(s, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad l, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{table}), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-l}))), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{lr-free-list-nodes}\,(\textit{max-addr}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(\text{lr-apply-subr}\,(\text{s->lr1}\,(s, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad l, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{table}), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-l}))), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(\text{lr-apply-subr}\,(\text{s->lr1}\,(s, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad l, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{table}), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{new-l})))) \\
 & \quad = \quad \text{lr-count-free-nodes}\,(\text{fetch}\,(\text{identity}\,(\textsc{lr-fp-addr}), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(l)), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{lr-free-list-nodes}\,(\textit{max-addr}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(l)), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{p-data-segment}\,(l))) \ \textbf{endlet}
\end{array}
$$

THEOREM: lr-eval-s-eval-equivalence-lr-check-result-flag-list
let *lr-eval* **be** lr-eval ('list,
lr-set-pos (s->lr1 (*s*, *l*, *table*), dv (s-pos (*s*), 1)),
c)
in
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ lr-proper-heapp (p-data-segment (*l*))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ lr-s-similar-statesp (s-params (*s*),
s-temps (*s*),
s->lr1 (*s*, *l*, *table*),
table)
∧ s-good-statep (*s*, *c*)
∧ (p-psw (*lr-eval*) = 'run)
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (car (s-expr (*s*)) ≠ S-TEMP-EVAL)
∧ (car (s-expr (*s*)) ≠ S-TEMP-TEST)
∧ (car (s-expr (*s*)) ≠ S-TEMP-FETCH)
∧ (car (s-expr (*s*)) ≠ 'quote))
→ lr-check-result ('list,
s-ans (s-eval ('list,
s-set-pos (*s*, dv (s-pos (*s*), 1)),
c)),
p-temp-stk (*lr-eval*),
p-data-segment (*lr-eval*),
p-temp-stk (*l*)) **endlet**

THEOREM: cadddr-max-r
cadddr (max-r (*list1*, *list2*)) = (cadddr (*list1*) + cadddr (*list2*))

THEOREM: lr-eval-s-eval-heap-r-lr-count-lr-free-list-nodes
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ lr-proper-heapp (p-data-segment (*l*))
∧ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
∧ s-good-statep (*s*, *c*)
∧ (p-psw (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)) = 'run)
∧ (s-err-flag (s-eval (*flag*, *s*, *c*)) = 'run))
→ ((lr-count-free-nodes (fetch (LR-FP-ADDR,
p-data-segment (lr-eval (*flag*,
s->lr1 (*s*, *l*, *table*),

c))),
lr-free-list-nodes (lr-max-node (p-data-segment (l)),
p-data-segment (lr-eval (*flag*,
s->lr1 (s,
l,
table),
c))),
p-data-segment (lr-eval (*flag*, s->lr1 (s, l, *table*), c)))
+ s-eval-heap-r (*flag*, s, c))
= lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (l)),
lr-free-list-nodes (lr-max-node (p-data-segment (l)),
p-data-segment (l)),
p-data-segment (l)))

THEOREM: lr-check-resourcesp-list-set-expr-nx
(listp (s-pos (s))
∧ listp (s-expr-list (s))
∧ good-posp (`'list`, s-pos (s), s-body (s-prog (s)))
∧ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
∧ lr-proper-heapp (p-data-segment (l))
∧ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
∧ lr-s-similar-statesp (s-params (s), s-temps (s), s->lr1 (s, l, *table*), *table*)
∧ s-good-statep (s, c)
∧ (p-psw (lr-eval (**t**, s->lr1 (s, l, *table*), c)) = `'run`)
∧ (s-err-flag (s-eval (**t**, s, c)) = `'run`)
∧ lr-check-resourcesp (`'list`, s, l, c))
→ lr-check-resourcesp (`'list`,
s-set-expr (s-eval (**t**, s, c), s, nx (s-pos (s))),
lr-eval (**t**, s->lr1 (s, l, *table*), c),
c)

THEOREM: lr-check-resourcesp-lr-push-tstk-flag-run
(lr-check-resourcesp (*flag*, s, l, c)
∧ (*flag* ≠ `'list`)
∧ litatom (s-expr (s))
∧ (c ≉ `0`)
∧ (s-err-flag (s) = `'run`))
→ (p-psw (lr-push-tstk (s->lr1 (s, l, *table*),
cdr (assoc (s-expr (s), bindings (car (p-ctrl-stk (l)))))))
= `'run`)

THEOREM: lr-check-resourcesp-s-set-pos-if-cadr
(lr-check-resourcesp (*flag*, s, l, c)
∧ s-good-statep (s, c)
∧ (*flag* ≠ `'list`)

$\wedge$ $(c \not\simeq \texttt{0})$
$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) = `'if`))
$\rightarrow$ lr-check-resourcesp (**t**, s-set-pos (s, dv (s-pos (s), `1`)), l, c)

THEOREM: s-eval-subsetp-s-collect-temp-alist-s-set-pos-if
(listp (s-expr (s))
$\wedge$ (car (s-expr (s)) = `'if`)
$\wedge$ s-good-statep (s, c)
$\wedge$ good-posp1 (dv (s-pos (s), `1`), s-body (s-prog (s)))
$\wedge$ s-check-temps-setp (s-temps (s))
$\wedge$ s-all-temps-setp (**t**, cadr (s-expr (s)), temp-alist-to-set (s-temps (s)))
$\wedge$ s-all-progs-temps-setp (s-progs (s))
$\wedge$ (s-err-flag (s-eval (**t**, s-set-pos (s, dv (s-pos (s), `1`)), c)) = `'run`))
$\rightarrow$ subsetp (s-collect-all-temps (**t**, cadr (s-expr (s))),
temp-alist-to-set (s-temps (s-eval (**t**,
s-set-pos (s, dv (s-pos (s), `1`)),
c))))

THEOREM: length-p-temp-stk-lr-pop-tstk-lr-eval-flag-t
(proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ good-posp1 (*pos*, s-body (s-prog (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ s-good-statep (s, c)
$\wedge$ (p-psw (lr-if-ok (lr-eval (**t**, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c)))
= `'run`))
$\rightarrow$ (length (p-temp-stk (lr-pop-tstk (lr-if-ok (lr-eval (**t**,
lr-set-pos (s->lr1 (s,
l,
table),
pos),
c)))))
= length (p-temp-stk (l)))

THEOREM: lr-eval-s->lr1-preserves-p-ctrl-stk-size-lr-set-pos
(proper-p-statep (lr->p (l))
$\wedge$ good-posp (*flag*, *pos*, program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval (*flag*, lr-set-pos (l, *pos*), c)) = `'run`))
$\rightarrow$ (p-ctrl-stk-size (p-ctrl-stk (lr-eval (*flag*, lr-set-pos (l, *pos*), c)))
= p-ctrl-stk-size (p-ctrl-stk (l)))

THEOREM: lr-check-resourcesp-lr-pop-tstk-lr-eval-1
let *lr-eval* **be** lr-if-ok (lr-eval (**t**, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c))
in

$$
\begin{array}{ll}
((c \not\simeq \texttt{0}) & \\
\wedge & \text{listp}\,(\text{s-expr}\,(s)) \\
\wedge & (\text{car}\,(\text{s-expr}\,(s)) = \texttt{'if}) \\
\wedge & \text{proper-p-statep}\,(\text{lr->p}\,(\text{s->lr1}\,(s,\ l,\ \mathit{table}))) \\
\wedge & \text{good-posp1}\,(\text{s-pos}\,(s),\ \text{s-body}\,(\text{s-prog}\,(s))) \\
\wedge & \text{lr-proper-heapp}\,(\text{p-data-segment}\,(l)) \\
\wedge & \text{lr-programs-properp}\,(\text{s->lr1}\,(s,\ l,\ \mathit{table}),\ \mathit{table}) \\
\wedge & \text{lr-s-similar-statesp}\,(\text{s-params}\,(s), \\
 & \qquad\qquad \text{s-temps}\,(s), \\
 & \qquad\qquad \text{s->lr1}\,(s,\ l,\ \mathit{table}), \\
 & \qquad\qquad \mathit{table}) \\
\wedge & \text{s-good-statep}\,(s,\ c) \\
\wedge & (\text{p-psw}\,(\mathit{lr\text{-}eval}) = \texttt{'run}) \\
\wedge & (\mathit{pos} = \text{dv}\,(\text{s-pos}\,(s),\ \texttt{1})) \\
\wedge & (\text{s-err-flag}\,(\text{s-eval}\,(\mathbf{t},\ \text{s-set-pos}\,(s,\ \mathit{pos}),\ c)) = \texttt{'run}) \\
\wedge & \text{s-ans}\,(\text{s-eval}\,(\mathbf{t},\ \text{s-set-pos}\,(s,\ \text{dv}\,(\text{s-pos}\,(s),\ \texttt{1})),\ c)) \\
\wedge & \text{lr-check-resourcesp}\,(\mathit{flag},\ s,\ l,\ c) \\
\wedge & (\mathit{flag} \neq \texttt{'list})) \\
\rightarrow & \text{lr-check-resourcesp}\,(\mathbf{t}, \\
 & \qquad\qquad \text{s-set-expr}\,(\text{s-eval}\,(\mathbf{t},\ \text{s-set-pos}\,(s,\ \mathit{pos}),\ c), \\
 & \qquad\qquad\qquad\qquad s, \\
 & \qquad\qquad\qquad\qquad \text{dv}\,(\text{s-pos}\,(s),\ \texttt{2})), \\
 & \qquad\qquad \text{lr-pop-tstk}\,(\mathit{lr\text{-}eval}), \\
 & \qquad\qquad c)\ \textbf{endlet}
\end{array}
$$

THEOREM: lr-check-resourcesp-lr-pop-tstk-lr-eval-2

$$
\begin{array}{ll}
\textbf{let}\ \mathit{lr\text{-}eval}\ \ \textbf{be}\ \ \text{lr-if-ok}\,(\text{lr-eval}\,(\mathbf{t},\ \text{lr-set-pos}\,(\text{s->lr1}\,(s,\ l,\ \mathit{table}),\ \mathit{pos}),\ c)) & \\
\textbf{in} & \\
((c \not\simeq \texttt{0}) & \\
\wedge & \text{listp}\,(\text{s-expr}\,(s)) \\
\wedge & (\text{car}\,(\text{s-expr}\,(s)) = \texttt{'if}) \\
\wedge & \text{proper-p-statep}\,(\text{lr->p}\,(\text{s->lr1}\,(s,\ l,\ \mathit{table}))) \\
\wedge & \text{good-posp1}\,(\text{s-pos}\,(s),\ \text{s-body}\,(\text{s-prog}\,(s))) \\
\wedge & \text{lr-proper-heapp}\,(\text{p-data-segment}\,(l)) \\
\wedge & \text{lr-programs-properp}\,(\text{s->lr1}\,(s,\ l,\ \mathit{table}),\ \mathit{table}) \\
\wedge & \text{lr-s-similar-statesp}\,(\text{s-params}\,(s), \\
 & \qquad\qquad \text{s-temps}\,(s), \\
 & \qquad\qquad \text{s->lr1}\,(s,\ l,\ \mathit{table}), \\
 & \qquad\qquad \mathit{table}) \\
\wedge & \text{s-good-statep}\,(s,\ c) \\
\wedge & (\text{p-psw}\,(\mathit{lr\text{-}eval}) = \texttt{'run}) \\
\wedge & (\mathit{pos} = \text{dv}\,(\text{s-pos}\,(s),\ \texttt{1})) \\
\wedge & (\text{s-err-flag}\,(\text{s-eval}\,(\mathbf{t},\ \text{s-set-pos}\,(s,\ \mathit{pos}),\ c)) = \texttt{'run}) \\
\wedge & (\neg\ \text{s-ans}\,(\text{s-eval}\,(\mathbf{t},\ \text{s-set-pos}\,(s,\ \text{dv}\,(\text{s-pos}\,(s),\ \texttt{1})),\ c)))
\end{array}
$$

$\wedge$ lr-check-resourcesp (*flag*, s, l, c)
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ lr-check-resourcesp (**t**,
s-set-expr (s-eval (**t**, s-set-pos (s, *pos*), c),
s,
dv (s-pos (s), 3)),
lr-pop-tstk (*lr-eval*),
c) **endlet**

THEOREM: lr-check-resourcesp-s-temp-eval
(($c \not\simeq$ 0)
$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) = S-TEMP-EVAL)
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ (*pos* = dv (s-pos (s), 1))
$\wedge$ lr-check-resourcesp (*flag*, s, l, c)
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ lr-check-resourcesp (**t**, s-set-pos (s, *pos*), l, c)

THEOREM: lr-check-resourcesp-s-temp-test
(($c \not\simeq$ 0)
$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) = S-TEMP-TEST)
$\wedge$ ($\neg$ s-temp-setp (cadr (s-expr (s)), s-temps (s)))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ (*pos* = dv (s-pos (s), 1))
$\wedge$ lr-check-resourcesp (*flag*, s, l, c)
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ lr-check-resourcesp (**t**, s-set-pos (s, *pos*), l, c)

THEOREM: lr-do-temp-fetch-lr-check-resourcesp-temp-test
(lr-check-resourcesp (*flag*, s, l, c)
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ listp (s-expr (s))
$\wedge$ ((car (s-expr (s)) = S-TEMP-TEST)
$\vee$ (car (s-expr (s)) = S-TEMP-FETCH))
$\wedge$ s-good-statep (s, c)
$\wedge$ lr-s-similar-statesp (s-params (s), s-temps (s), s->lr1 (s, l, *table*), *table*)
$\wedge$ ($c \not\simeq$ 0)
$\wedge$ s-temp-setp (cadr (s-expr (s)), s-temps (s))
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ (p-psw (lr-do-temp-fetch (s->lr1 (s, l, *table*))) = 'run)

THEOREM: lr-push-tstk-lr-check-resourcesp-quote
(lr-check-resourcesp (*flag*, *s*, *l*, *c*)
 ∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
 ∧ listp (s-expr (*s*))
 ∧ (car (s-expr (*s*)) = 'quote)
 ∧ s-good-statep (*s*, *c*)
 ∧ (*c* ≠ 0)
 ∧ (*flag* ≠ 'list))
→ (p-psw (lr-push-tstk (s->lr1 (*s*, *l*, *table*),
 cadr (lr-expr (s->lr1 (*s*, *l*, *table*)))))
 = 'run)

THEOREM: lr-check-resourcesp-funcall
((*c* ≠ 0)
 ∧ listp (s-expr (*s*))
 ∧ (car (s-expr (*s*)) ≠ 'if)
 ∧ (car (s-expr (*s*)) ≠ S-TEMP-EVAL)
 ∧ (car (s-expr (*s*)) ≠ S-TEMP-TEST)
 ∧ (car (s-expr (*s*)) ≠ S-TEMP-FETCH)
 ∧ (car (s-expr (*s*)) ≠ 'quote)
 ∧ (s-err-flag (s-eval ('list, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*)) = 'run)
 ∧ proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ s-good-statep (*s*, *c*)
 ∧ lr-check-resourcesp (*flag*, *s*, *l*, *c*)
 ∧ (*flag* ≠ 'list))
→ lr-check-resourcesp ('list, s-set-pos (*s*, dv (s-pos (*s*), 1)), *l*, *c*)

THEOREM: numberp-s-eval-temp-ctrl-ws-heap-r-opened
(car (s-eval-r (*flag*, *s*, *c*)) ∈ **N**)
∧ (cadr (s-eval-r (*flag*, *s*, *c*)) ∈ **N**)
∧ (caddr (s-eval-r (*flag*, *s*, *c*)) ∈ **N**)
∧ (cadddr (s-eval-r (*flag*, *s*, *c*)) ∈ **N**)

THEOREM: lessp-1-not-zerop-exp
((*m* ≠ 0) ∧ (1 < *n*)) → (1 < exp (*n*, *m*))

THEOREM: lessp-1-not-zerop-log
((1 < *c*) ∧ (*n* ∈ **N**)) → ((log (*c*, *n*) < 1) = (*n* < 1))

DEFINITION:
induct-hint-18 (*c*, *n*, *m*)
= **if** *c* < 2 **then t**

elseif $n \simeq \texttt{0}$ **then t**
elseif $m \simeq \texttt{0}$ **then t**
else induct-hint-18 $(c,\ n \div c,\ m - 1)$ **endif**

THEOREM: times-quotient-lessp-fact-1
$((c \not\simeq \texttt{0}) \wedge (n \in \mathbf{N}) \wedge (m \in \mathbf{N}))$
$\rightarrow \quad ((n < (c * m)) = ((n \div c) < m))$

THEOREM: exp-log-lessp-fact-1
$((\texttt{1} < c) \wedge (n \in \mathbf{N}) \wedge (m \in \mathbf{N}))$
$\rightarrow \quad ((n < \exp(c,\ m)) = (\log(c,\ n) < (1 + m)))$

EVENT: Disable times-quotient-lessp-fact-1.

THEOREM: adpp-untag-add-addr-offset-car
(lr-good-pointerp (*addr*, *data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ (untag (fetch (*addr*, *data-seg*)) = LR-CONS-TAG)
$\wedge$ lr-proper-heapp (*data-seg*))
$\rightarrow$ adpp (untag (add-addr (*addr*, identity (LR-CAR-OFFSET))), *data-seg*)

THEOREM: adpp-untag-add-addr-offset-cdr
(lr-good-pointerp (*addr*, *data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ (untag (fetch (*addr*, *data-seg*)) = LR-CONS-TAG)
$\wedge$ lr-proper-heapp (*data-seg*))
$\rightarrow$ adpp (untag (add-addr (*addr*, identity (LR-CDR-OFFSET))), *data-seg*)

THEOREM: exp-log-2-lessp-add1-fact-1
$((1 + n) < \exp(\texttt{2},\ m)) = (\log(\texttt{2},\ 1 + n) < (1 + m))$

```
;; The P-TEST-BOOL-AND-JUMP cause a lot of case splits after being opened
;; and the result rewritten with P-OBJECTP-TYPE, so we prove two simple
;; lemmas and disable it, this should hopefully speed up the proof.
```

THEOREM: p-test-bool-and-jump-okp-t-cons-bool-t
p-test-bool-and-jump-okp (list (*ins-name*, 't, *label*),
p-state (*pc*,
ctrl-stk,
cons ('(bool t), *temp-stk*),
prog-seg,
data-seg,
max-ctrl,
max-temp,
word-size,
psw))
$=$ **t**

THEOREM: p-test-bool-and-jump-okp-f-cons-bool-t
p-test-bool-and-jump-okp (list (*ins-name*, `'f`, *label*),
p-state (*pc*,
ctrl-stk,
cons (`'(bool t)`, *temp-stk*),
prog-seg,
data-seg,
max-ctrl,
max-temp,
word-size,
psw))
= **t**

THEOREM: p-test-bool-and-jump-okp-t-cons-bool-f
p-test-bool-and-jump-okp (list (*ins-name*, `'t`, *label*),
p-state (*pc*,
ctrl-stk,
cons (`'(bool f)`, *temp-stk*),
prog-seg,
data-seg,
max-ctrl,
max-temp,
word-size,
psw))
= **t**

THEOREM: p-test-bool-and-jump-okp-f-cons-bool-f
p-test-bool-and-jump-okp (list (*ins-name*, `'f`, *label*),
p-state (*pc*,
ctrl-stk,
cons (`'(bool f)`, *temp-stk*),
prog-seg,
data-seg,
max-ctrl,
max-temp,
word-size,
psw))
= **t**

EVENT: Disable p-test-bool-and-jump-okp.

THEOREM: p-psw-run-run-car-lr-check-resourcesp
(lr-check-result (`'list`,
s-ans (*new-s*),

```
                p-temp-stk (new-l),
                p-data-segment (new-l),
                orig-temp-stk)
  ∧  proper-p-statep (lr->p (new-l))
  ∧  (p-psw (new-l) = 'run)
  ∧  lr-programs-properp (new-l, table)
  ∧  (unlabel (get (offset (pc),
                   program-body (assoc (area-name (pc),
                                        p-prog-segment (lr->p (new-l))))))
     =  '(call car))
  ∧  (p-max-temp-stk-size (new-l)
     ≮  (length (p-temp-stk (new-l))
        +  car (s-apply-subr-r ('car, new-s))))
  ∧  (p-max-ctrl-stk-size (new-l)
     ≮  (p-ctrl-stk-size (p-ctrl-stk (new-l))
        +  cadr (s-apply-subr-r ('car, new-s))))
  ∧  (p-word-size (new-l)
     ≮  max (S-MAX-SUBR-REQS, caddr (s-apply-subr-r ('car, new-s))))
  ∧  (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (new-l)),
                           lr-free-list-nodes (lr-max-node (p-data-segment (new-l)),
                                               p-data-segment (new-l)),
                           p-data-segment (new-l))
     ≮  cadddr (s-apply-subr-r ('car, new-s)))
  ∧  (length (p-temp-stk (new-l)) ≮ arity ('car))
  ∧  (length (s-ans (new-s)) = arity ('car)))
  →  (p-psw (p (p-set-pc (lr->p (new-l), pc),
                p-car-clock (p-set-pc (lr->p (new-l), pc))))
     =  'run)
```

THEOREM: p-psw-run-run-cdr-lr-check-resourcesp

```
 (lr-check-result ('list,
                   s-ans (new-s),
                   p-temp-stk (new-l),
                   p-data-segment (new-l),
                   orig-temp-stk)
  ∧  proper-p-statep (lr->p (new-l))
  ∧  (p-psw (new-l) = 'run)
  ∧  lr-programs-properp (new-l, table)
  ∧  (unlabel (get (offset (pc),
                   program-body (assoc (area-name (pc),
                                        p-prog-segment (lr->p (new-l))))))
     =  '(call cdr))
  ∧  (p-max-temp-stk-size (new-l)
     ≮  (length (p-temp-stk (new-l))
```

$$
\begin{array}{ll}
 & \quad + \quad \text{car}\,(\text{s-apply-subr-r}\,(\texttt{'cdr},\ \textit{new-s}))))\\
\wedge & (\text{p-max-ctrl-stk-size}\,(\textit{new-l})\\
 & \quad \not< \quad (\text{p-ctrl-stk-size}\,(\text{p-ctrl-stk}\,(\textit{new-l}))\\
 & \qquad + \quad \text{cadr}\,(\text{s-apply-subr-r}\,(\texttt{'cdr},\ \textit{new-s}))))\\
\wedge & (\text{p-word-size}\,(\textit{new-l})\\
 & \quad \not< \quad \max\,(\text{S-MAX-SUBR-REQS},\ \text{caddr}\,(\text{s-apply-subr-r}\,(\texttt{'cdr},\ \textit{new-s}))))\\
\wedge & (\text{lr-count-free-nodes}\,(\text{fetch}\,(\text{LR-FP-ADDR},\ \text{p-data-segment}\,(\textit{new-l})),\\
 & \qquad\qquad\qquad\qquad \text{lr-free-list-nodes}\,(\text{lr-max-node}\,(\text{p-data-segment}\,(\textit{new-l})),\\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{p-data-segment}\,(\textit{new-l})),\\
 & \qquad\qquad\qquad\qquad \text{p-data-segment}\,(\textit{new-l}))\\
 & \quad \not< \quad \text{cadddr}\,(\text{s-apply-subr-r}\,(\texttt{'cdr},\ \textit{new-s})))\\
\wedge & (\text{length}\,(\text{p-temp-stk}\,(\textit{new-l})) \not< \text{arity}\,(\texttt{'cdr}))\\
\wedge & (\text{length}\,(\text{s-ans}\,(\textit{new-s})) = \text{arity}\,(\texttt{'cdr})))\\
\rightarrow & (\text{p-psw}\,(\text{p}\,(\text{p-set-pc}\,(\text{lr->p}\,(\textit{new-l}),\ \textit{pc}),\\
 & \qquad\qquad \text{p-cdr-clock}\,(\text{p-set-pc}\,(\text{lr->p}\,(\textit{new-l}),\ \textit{pc}))))\\
 & \quad = \quad \texttt{'run})
\end{array}
$$

THEOREM: lessp-plus-remainder-0-fact

$$
\begin{array}{ll}
 & (((\textit{offset1}\ \textbf{mod}\ \textit{max}) = \texttt{0})\\
\wedge & ((\textit{offset2}\ \textbf{mod}\ \textit{max}) = \texttt{0})\\
\wedge & (n < \textit{max})\\
\wedge & (\textit{offset1} \in \mathbf{N})\\
\wedge & (\textit{offset2} \in \mathbf{N}))\\
\rightarrow & (((n + \textit{offset1}) < \textit{offset2}) = (\textit{offset1} < \textit{offset2}))
\end{array}
$$

THEOREM: lr-boundary-nodep-lessp-plus-fact

$$
\begin{array}{ll}
 & (\text{lr-boundary-nodep}\,(\textit{addr1})\\
\wedge & \text{lr-boundary-nodep}\,(\textit{addr2})\\
\wedge & (n < \text{LR-NODE-SIZE})\\
\wedge & (\text{offset}\,(\textit{addr1}) \in \mathbf{N})\\
\wedge & (\text{offset}\,(\textit{addr2}) \in \mathbf{N}))\\
\rightarrow & (((n + \text{offset}\,(\textit{addr1})) < \text{offset}\,(\textit{addr2}))\\
 & \quad = \quad (\text{offset}\,(\textit{addr1}) < \text{offset}\,(\textit{addr2})))
\end{array}
$$

THEOREM: adpp-untag-add-addr-lr-nodep-not-max-addr

$$
\begin{array}{ll}
 & (\text{adpp}\,(\text{untag}\,(\textit{addr}),\ \textit{data-seg})\\
\wedge & \text{lr-boundary-nodep}\,(\textit{addr})\\
\wedge & (\text{area-name}\,(\textit{addr}) = \texttt{'heap})\\
\wedge & \text{definedp}\,(\text{LR-HEAP-NAME},\ \textit{data-seg})\\
\wedge & \text{lr-boundary-nodep}\,(\text{lr-max-node}\,(\textit{data-seg}))\\
\wedge & (\text{offset}\,(\textit{addr}) < \text{offset}\,(\text{lr-max-node}\,(\textit{data-seg})))\\
\wedge & (n < \text{LR-NODE-SIZE}))\\
\rightarrow & \text{adpp}\,(\text{untag}\,(\text{add-addr}\,(\textit{addr},\ n)),\ \textit{data-seg})
\end{array}
$$

THEOREM: adpp-untag-add-addr-offset-on-free-listp

(lr-proper-p-areasp (*data-seg*)
 ∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
 lr-free-list-nodes (lr-max-node (*data-seg*),
 data-seg),
 data-seg)
 ≮ 1)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ (*n* < LR-NODE-SIZE))
→ adpp (untag (add-addr (fetch (identity (LR-FP-ADDR), *data-seg*), *n*)), *data-seg*)

THEOREM: p-psw-run-run-cons-lr-check-resourcesp
(lr-check-result ('list,
 s-ans (*new-s*),
 p-temp-stk (*new-l*),
 p-data-segment (*new-l*),
 orig-temp-stk)
 ∧ proper-p-statep (lr->p (*new-l*))
 ∧ (p-psw (*new-l*) = 'run)
 ∧ lr-programs-properp (*new-l*, *table*)
 ∧ (unlabel (get (offset (*pc*),
 program-body (assoc (area-name (*pc*),
 p-prog-segment (lr->p (*new-l*))))))
 = '(call cons))
 ∧ (p-max-temp-stk-size (*new-l*)
 ≮ (length (p-temp-stk (*new-l*))
 + car (s-apply-subr-r ('cons, *new-s*))))
 ∧ (p-max-ctrl-stk-size (*new-l*)
 ≮ (p-ctrl-stk-size (p-ctrl-stk (*new-l*))
 + cadr (s-apply-subr-r ('cons, *new-s*))))
 ∧ (p-word-size (*new-l*)
 ≮ max (S-MAX-SUBR-REQS, caddr (s-apply-subr-r ('cons, *new-s*))))
 ∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (*new-l*)),
 lr-free-list-nodes (lr-max-node (p-data-segment (*new-l*)),
 p-data-segment (*new-l*)),
 p-data-segment (*new-l*))
 ≮ cadddr (s-apply-subr-r ('cons, *new-s*)))
 ∧ (length (p-temp-stk (*new-l*)) ≮ arity ('cons))
 ∧ (length (s-ans (*new-s*)) = arity ('cons)))
→ (p-psw (p (p-set-pc (lr->p (*new-l*), *pc*),
 p-cons-clock (p-set-pc (lr->p (*new-l*), *pc*))))
 = 'run)

THEOREM: p-psw-run-run-false-lr-check-resourcesp
(lr-check-result ('list,

```
                s-ans (new-s),
                p-temp-stk (new-l),
                p-data-segment (new-l),
                orig-temp-stk)
∧   proper-p-statep (lr->p (new-l))
∧   (p-psw (new-l) = 'run)
∧   lr-programs-properp (new-l, table)
∧   (unlabel (get (offset (pc),
                  program-body (assoc (area-name (pc),
                                       p-prog-segment (lr->p (new-l))))))
     =  '(call false))
∧   (p-max-temp-stk-size (new-l)
     ≮  (length (p-temp-stk (new-l))
         +  car (s-apply-subr-r ('false, new-s))))
∧   (p-max-ctrl-stk-size (new-l)
     ≮  (p-ctrl-stk-size (p-ctrl-stk (new-l))
         +  cadr (s-apply-subr-r ('false, new-s))))
∧   (p-word-size (new-l)
     ≮  max (S-MAX-SUBR-REQS, caddr (s-apply-subr-r ('false, new-s))))
∧   (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (new-l)),
                          lr-free-list-nodes (lr-max-node (p-data-segment (new-l)),
                                              p-data-segment (new-l)),
                          p-data-segment (new-l))
     ≮  cadddr (s-apply-subr-r ('false, new-s)))
∧   (length (p-temp-stk (new-l)) ≮ arity ('false))
∧   (length (s-ans (new-s)) = arity ('false)))
→   (p-psw (p (p-set-pc (lr->p (new-l), pc),
               p-false-clock (p-set-pc (lr->p (new-l), pc))))
     =  'run)
```

THEOREM: p-psw-run-run-falsep-lr-check-resourcesp

```
(lr-check-result ('list,
                  s-ans (new-s),
                  p-temp-stk (new-l),
                  p-data-segment (new-l),
                  orig-temp-stk)
∧   proper-p-statep (lr->p (new-l))
∧   (p-psw (new-l) = 'run)
∧   lr-programs-properp (new-l, table)
∧   (unlabel (get (offset (pc),
                  program-body (assoc (area-name (pc),
                                       p-prog-segment (lr->p (new-l))))))
     =  '(call falsep))
∧   (p-max-temp-stk-size (new-l)
```

$\not<$ (length (p-temp-stk (*new-l*))
+ car (s-apply-subr-r (`'falsep`, *new-s*))))
$\wedge$ (p-max-ctrl-stk-size (*new-l*)
$\not<$ (p-ctrl-stk-size (p-ctrl-stk (*new-l*))
+ cadr (s-apply-subr-r (`'falsep`, *new-s*))))
$\wedge$ (p-word-size (*new-l*)
$\not<$ max (S-MAX-SUBR-REQS, caddr (s-apply-subr-r (`'falsep`, *new-s*))))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (*new-l*)),
lr-free-list-nodes (lr-max-node (p-data-segment (*new-l*)),
p-data-segment (*new-l*)),
p-data-segment (*new-l*))
$\not<$ cadddr (s-apply-subr-r (`'falsep`, *new-s*)))
$\wedge$ (length (p-temp-stk (*new-l*)) $\not<$ arity (`'falsep`))
$\wedge$ (length (s-ans (*new-s*)) = arity (`'falsep`)))
$\rightarrow$ (p-psw (p (p-set-pc (lr->p (*new-l*), *pc*),
p-falsep-clock (p-set-pc (lr->p (*new-l*), *pc*))))
= `'run`)

THEOREM: p-psw-run-run-listp-lr-check-resourcesp
(lr-check-result (`'list`,
s-ans (*new-s*),
p-temp-stk (*new-l*),
p-data-segment (*new-l*),
orig-temp-stk)
$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ lr-programs-properp (*new-l*, *table*)
$\wedge$ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (*new-l*))))))
= `'(call listp)`)
$\wedge$ (p-max-temp-stk-size (*new-l*)
$\not<$ (length (p-temp-stk (*new-l*))
+ car (s-apply-subr-r (`'listp`, *new-s*))))
$\wedge$ (p-max-ctrl-stk-size (*new-l*)
$\not<$ (p-ctrl-stk-size (p-ctrl-stk (*new-l*))
+ cadr (s-apply-subr-r (`'listp`, *new-s*))))
$\wedge$ (p-word-size (*new-l*)
$\not<$ max (S-MAX-SUBR-REQS, caddr (s-apply-subr-r (`'listp`, *new-s*))))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (*new-l*)),
lr-free-list-nodes (lr-max-node (p-data-segment (*new-l*)),
p-data-segment (*new-l*)),
p-data-segment (*new-l*))
$\not<$ cadddr (s-apply-subr-r (`'listp`, *new-s*)))

$\wedge$ (length (p-temp-stk (*new-l*)) $\not<$ arity (`'listp`))
$\wedge$ (length (s-ans (*new-s*)) = arity (`'listp`)))
$\rightarrow$ (p-psw (p (p-set-pc (lr->p (*new-l*), *pc*),
p-listp-clock (p-set-pc (lr->p (*new-l*), *pc*))))
= `'run`)

THEOREM: p-psw-run-run-nlistp-lr-check-resourcesp
(lr-check-result (`'list`,
s-ans (*new-s*),
p-temp-stk (*new-l*),
p-data-segment (*new-l*),
orig-temp-stk)
$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ lr-programs-properp (*new-l*, *table*)
$\wedge$ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (*new-l*))))))
= `'(call nlistp)`)
$\wedge$ (p-max-temp-stk-size (*new-l*)
$\not<$ (length (p-temp-stk (*new-l*))
+ car (s-apply-subr-r (`'nlistp`, *new-s*))))
$\wedge$ (p-max-ctrl-stk-size (*new-l*)
$\not<$ (p-ctrl-stk-size (p-ctrl-stk (*new-l*))
+ cadr (s-apply-subr-r (`'nlistp`, *new-s*))))
$\wedge$ (p-word-size (*new-l*)
$\not<$ max (S-MAX-SUBR-REQS, caddr (s-apply-subr-r (`'nlistp`, *new-s*))))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (*new-l*)),
lr-free-list-nodes (lr-max-node (p-data-segment (*new-l*)),
p-data-segment (*new-l*)),
p-data-segment (*new-l*))
$\not<$ cadddr (s-apply-subr-r (`'nlistp`, *new-s*)))
$\wedge$ (length (p-temp-stk (*new-l*)) $\not<$ arity (`'nlistp`))
$\wedge$ (length (s-ans (*new-s*)) = arity (`'nlistp`)))
$\rightarrow$ (p-psw (p (p-set-pc (lr->p (*new-l*), *pc*),
p-nlistp-clock (p-set-pc (lr->p (*new-l*), *pc*))))
= `'run`)

THEOREM: p-psw-run-run-true-lr-check-resourcesp
(lr-check-result (`'list`,
s-ans (*new-s*),
p-temp-stk (*new-l*),
p-data-segment (*new-l*),
orig-temp-stk)

$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ lr-programs-properp (*new-l*, *table*)
$\wedge$ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (*new-l*))))))
= `'(call true)`)
$\wedge$ (p-max-temp-stk-size (*new-l*)
$\not<$ (length (p-temp-stk (*new-l*))
+ car (s-apply-subr-r (`'true`, *new-s*))))
$\wedge$ (p-max-ctrl-stk-size (*new-l*)
$\not<$ (p-ctrl-stk-size (p-ctrl-stk (*new-l*))
+ cadr (s-apply-subr-r (`'true`, *new-s*))))
$\wedge$ (p-word-size (*new-l*)
$\not<$ max (S-MAX-SUBR-REQS, caddr (s-apply-subr-r (`'true`, *new-s*))))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (*new-l*)),
lr-free-list-nodes (lr-max-node (p-data-segment (*new-l*)),
p-data-segment (*new-l*)),
p-data-segment (*new-l*))
$\not<$ cadddr (s-apply-subr-r (`'true`, *new-s*)))
$\wedge$ (length (p-temp-stk (*new-l*)) $\not<$ arity (`'true`))
$\wedge$ (length (s-ans (*new-s*)) = arity (`'true`)))
$\rightarrow$ (p-psw (p (p-set-pc (lr->p (*new-l*), *pc*),
p-true-clock (p-set-pc (lr->p (*new-l*), *pc*))))
= `'run`)

THEOREM: p-psw-run-run-truep-lr-check-resourcesp
(lr-check-result (`'list`,
s-ans (*new-s*),
p-temp-stk (*new-l*),
p-data-segment (*new-l*),
orig-temp-stk)
$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ lr-programs-properp (*new-l*, *table*)
$\wedge$ (unlabel (get (offset (*pc*),
program-body (assoc (area-name (*pc*),
p-prog-segment (lr->p (*new-l*))))))
= `'(call truep)`)
$\wedge$ (p-max-temp-stk-size (*new-l*)
$\not<$ (length (p-temp-stk (*new-l*))
+ car (s-apply-subr-r (`'truep`, *new-s*))))
$\wedge$ (p-max-ctrl-stk-size (*new-l*)
$\not<$ (p-ctrl-stk-size (p-ctrl-stk (*new-l*))

$+$ cadr (s-apply-subr-r (`'truep`, *new-s*))))
$\wedge$ (p-word-size (*new-l*)
$\not<$ max (S-MAX-SUBR-REQS, caddr (s-apply-subr-r (`'truep`, *new-s*))))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (*new-l*)),
lr-free-list-nodes (lr-max-node (p-data-segment (*new-l*)),
p-data-segment (*new-l*)),
p-data-segment (*new-l*))
$\not<$ cadddr (s-apply-subr-r (`'truep`, *new-s*)))
$\wedge$ (length (p-temp-stk (*new-l*)) $\not<$ arity (`'truep`))
$\wedge$ (length (s-ans (*new-s*)) = arity (`'truep`)))
$\rightarrow$ (p-psw (p (p-set-pc (lr->p (*new-l*), *pc*),
p-truep-clock (p-set-pc (lr->p (*new-l*), *pc*))))
= `'run`)

THEOREM: length-last
listp (*l*) $\rightarrow$ (length (last (*l*)) = `1`)

THEOREM: equal-plus-lessp-fact
$((x + z) = y) \rightarrow ((y < (n + x)) = (z < n))$

THEOREM: not-lessp-lr-count-free-nodes-lr-eval-list-lr-set-pos
let *new-l* **be** lr-eval (`'list`, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)
in
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
$\wedge$ lr-proper-heapp (p-data-segment (*l*))
$\wedge$ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
$\wedge$ lr-s-similar-statesp (s-params (*s*),
s-temps (*s*),
s->lr1 (*s*, *l*, *table*),
table)
$\wedge$ s-good-statep (*s*, *c*)
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ (s-err-flag (s-eval (`'list`, s-set-pos (*s*, *pos*), *c*)) = `'run`)
$\wedge$ listp (s-expr (*s*))
$\wedge$ (car (s-expr (*s*)) $\neq$ `'if`)
$\wedge$ (car (s-expr (*s*)) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (s-expr (*s*)) $\neq$ S-TEMP-TEST)
$\wedge$ (car (s-expr (*s*)) $\neq$ S-TEMP-FETCH)
$\wedge$ (car (s-expr (*s*)) $\neq$ `'quote`)
$\wedge$ (*pos* = dv (s-pos (*s*), `1`))
$\wedge$ (*max-addr* = lr-max-node (p-data-segment (*l*))))
$\rightarrow$ ((lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
p-data-segment (*new-l*)),
lr-free-list-nodes (*max-addr*,

p-data-segment (*new-l*)),
p-data-segment (*new-l*))
< *n*)
= (lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
p-data-segment (*l*)),
lr-free-list-nodes (*max-addr*,
p-data-segment (*l*)),
p-data-segment (*l*))
< (s-eval-heap-r (`'list`,
s-set-pos (*s*, dv (s-pos (*s*), `1`)),
c)
+ *n*))) **endlet**

EVENT: Disable equal-plus-lessp-fact.

THEOREM: lr-programs-properp-definedp-subrp-runtime-support
((¬ definedp (car (lr-expr (*l*)), P-RUNTIME-SUPPORT-PROGRAMS))
∧ (car (lr-expr (*l*)) ≠ `'if`)
∧ subrp (car (lr-expr (*l*)))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*))))
→ (¬ lr-programs-properp (*l*, *table*))

THEOREM: p-psw-run-p-run-subr-lr-check-resourcesp
let *new-l* **be** lr-eval (`'list`, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*),
new-s **be** s-eval (`'list`, s-set-pos (*s*, *pos*), *c*),
pc **be** lr-return-pc (s->lr1 (*s*, *l*, *table*)),
r **be** s-apply-subr-r (car (s-expr (*s*)), s-eval (`'list`, s-set-pos (*s*, *pos*), *c*))
in
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ (*c* ≉ `0`)
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ `'if`)
∧ (car (s-expr (*s*)) ≠ `'quote`)
∧ (s-err-flag (*new-s*) = `'run`)
∧ subrp (car (s-expr (*s*)))
∧ (p-psw (*new-l*) = `'run`)
∧ lr-proper-heapp (p-data-segment (*l*))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ lr-s-similar-statesp (s-params (*s*),
s-temps (*s*),
s->lr1 (*s*, *l*, *table*),
table)
∧ s-good-statep (*s*, *c*)

$\wedge$ (p-max-temp-stk-size (l)
  $\not<$ (length (p-temp-stk (l))
    + arity (car (s-expr (s)))
    + car (r)))
$\wedge$ (p-max-ctrl-stk-size (l)
  $\not<$ (p-ctrl-stk-size (p-ctrl-stk (l)) + cadr (r)))
$\wedge$ (p-word-size (l) $\not<$ max (S-MAX-SUBR-REQS, caddr (r)))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, p-data-segment (l)),
    lr-free-list-nodes (lr-max-node (p-data-segment (l)),
      p-data-segment (l)),
    p-data-segment (l))
  $\not<$ (cadddr (r)
    + s-eval-heap-r ('list, s-set-pos (s, pos), c)))
$\wedge$ (pos = dv (s-pos (s), 1)))
$\rightarrow$ (p-psw (p-run-subr (car (s-expr (s)), p-set-pc (lr->p (new-l), pc)))
  = 'run) **endlet**

EVENT: Disable lr-programs-properp-definedp-subrp-runtime-support.

THEOREM: not-lessp-help-fact
$((x \not< y) \wedge (x \not< z)) \rightarrow ((x < \max(y, z)) = \mathbf{f})$

THEOREM: p-psw-run-lr-apply-subr-lr-check-resourcesp
($(c \not\simeq 0)$
$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ (s-err-flag (s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)) = 'run)
$\wedge$ subrp (car (s-expr (s)))
$\wedge$ (p-psw (lr-eval ('list,
    lr-set-pos (s->lr1 (s, l, $table$), dv (s-pos (s), 1)),
    c))
  = 'run)
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, $table$)))
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, $table$), $table$)
$\wedge$ lr-s-similar-statesp (s-params (s), s-temps (s), s->lr1 (s, l, $table$), $table$)
$\wedge$ s-good-statep (s, c)
$\wedge$ lr-check-resourcesp ($flag$, s, l, c)
$\wedge$ (p-word-size (l) $\not<$ S-MAX-SUBR-REQS)
$\wedge$ ($flag \neq$ 'list))
$\rightarrow$ (p-psw (lr-apply-subr (s->lr1 (s, l, $table$),
    lr-eval ('list,

lr-set-pos (s->lr1 (*s*, *l*, *table*),
dv (s-pos (*s*), 1)),
c)))
= 'run)

THEOREM: strip-logic-fnames-lr-compile-programs
strip-logic-fnames (lr-compile-programs (*programs*, *const-table*))
= strip-logic-fnames (*programs*)

THEOREM: strip-logic-fnames-cdr-lr-compile-programs
strip-logic-fnames (cdr (lr-compile-programs (*programs*, *const-table*)))
= strip-logic-fnames (cdr (*programs*))

THEOREM: lr-programs-properp-s->lr1-definedp-cdr-s-progs
(lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (car (s-expr (*s*)) ≠ 'quote)
∧ (¬ subrp (car (s-expr (*s*))))
∧ litatom (car (s-expr (*s*)))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ s-programs-okp (cdr (s-progs (*s*))))
→ definedp (user-fname (car (s-expr (*s*))), cdr (s-progs (*s*)))

THEOREM: s-programs-okp-formals-not-f
(s-programs-okp (*progs*) ∧ (*prog* ∈ *progs*)) → (s-formals (*prog*) ≠ **f**)

THEOREM: not-lessp-plus-arity-length-formals
(listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (car (s-expr (*s*)) ≠ 'quote)
∧ (¬ subrp (car (s-expr (*s*))))
∧ litatom (car (s-expr (*s*)))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ s-good-statep (*s*, *c*))
→ (((arity (car (s-expr (*s*))) + *x*) < length (formals (car (s-expr (*s*)))))
= **f**)

THEOREM: length-lr-make-temp-var-dcls
length (lr-make-temp-var-dcls (*temp-alist*)) = length (*temp-alist*)

THEOREM: length-lr-make-temp-name-alist-1
length (lr-make-temp-name-alist-1 (*initial*, *num-list*, *temp-list*, *formals*))
= length (*temp-list*)

THEOREM: length-lr-make-temp-name-alist
length (lr-make-temp-name-alist (*temp-list*, *formals*)) = length (*temp-list*)

THEOREM: p-ctrl-stk-size-0
(p-ctrl-stk-size (*ctrl-stk*) = `0`) = (¬ listp (*ctrl-stk*))

THEOREM: length-make-temps-entries
length (make-temps-entries (*list*)) = length (*list*)

THEOREM: s-eval-ctrl-r-funcall-opener
let *arg-s* **be** s-eval (`'list`, s-set-pos (*s*, dv (s-pos (*s*), `1`)), *c*)
in
((*c* ≠ `0`)
∧ s-good-statep (*s*, *c*)
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ `'if`)
∧ (car (s-expr (*s*)) ≠ `'quote`)
∧ (¬ subrp (car (s-expr (*s*))))
∧ litatom (car (s-expr (*s*)))
∧ (*flag* ≠ `'list`)
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ (s-err-flag (*arg-s*) = `'run`))
→ (s-eval-ctrl-r (*flag*, *s*, *c*)
= max (s-eval-ctrl-r (`'list`,
s-set-pos (*s*, dv (s-pos (*s*), `1`)),
c),
1 + (1 + (length (formals (car (s-expr (*s*))))
+ length (s-temp-list (assoc (user-fname (car (s-expr (*s*))),
s-progs (*s*))))
+ s-eval-ctrl-r (**t**,
s-fun-call-state (*arg-s*,
car (s-expr (*s*))),
c − 1))))) **endlet**

THEOREM: s-good-statep-formals-assoc-cdr-s-progs
(s-good-statep (*s*, *c*)
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ `'if`)
∧ (car (s-expr (*s*)) ≠ `'quote`)
∧ (¬ subrp (car (s-expr (*s*))))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ litatom (car (s-expr (*s*)))
∧ (*progs* = cdr (s-progs (*s*))))
→ (s-formals (assoc (user-fname (car (s-expr (*s*))), *progs*))
= formals (car (s-expr (*s*))))

THEOREM: not-lessp-p-ctrl-stk-size-make-p-call-frame
let *s-prog* **be** assoc (user-fname (car (s-expr (*s*))), cdr (s-progs (*s*))),
lr-eval **be** lr-eval ('list,
lr-set-pos (s->lr1 (*s*, *l*, *table*), dv (s-pos (*s*), 1)),
c)
in
(listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (car (s-expr (*s*)) ≠ 'quote)
∧ (¬ subrp (car (s-expr (*s*))))
∧ litatom (car (s-expr (*s*)))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ s-good-statep (*s*, *c*)
∧ (*c* ≉ 0)
∧ proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ (length (*temp-list*) = length (s-temp-list (*s-prog*)))
∧ (p-psw (*lr-eval*) = 'run)
∧ (s-err-flag (s-eval ('list, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*))
= 'run)
∧ (p-max-ctrl-stk-size (*l*)
⊀ (p-ctrl-stk-size (p-ctrl-stk (*l*))
\+ s-eval-ctrl-r (*flag*, *s*, *c*)))
∧ (*flag* ≠ 'list))
→ (p-max-ctrl-stk-size (*l*)
⊀ p-ctrl-stk-size (cons (make-p-call-frame (formals (car (s-expr (*s*))),
temp-stk,
temp-list,
pc),
p-ctrl-stk (*lr-eval*)))) **endlet**

THEOREM: definedp-0
definedp (*x*, 0) = **f**

THEOREM: not-definedp-user-fname-p-runtime-support-programs
¬ definedp (user-fname (*name*), P-RUNTIME-SUPPORT-PROGRAMS)

THEOREM: comp-programs-assoc-cons-opener
(user-fname (*name*) ≠ *prog1-name*)
→ (assoc (user-fname (*name*),
comp-programs (cons (cons (*prog1-name*, *prog1*), *progs*)))
= assoc (user-fname (*name*), comp-programs-1 (*progs*)))

THEOREM: lr-check-resourcesp-lr-funcall-p-psw-run
((*c* ≉ 0)

$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ ($\neg$ subrp (car (s-expr (s))))
$\wedge$ litatom (car (s-expr (s)))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ (p-psw (lr-eval ('list, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c))
= 'run)
$\wedge$ (s-err-flag (s-eval ('list, s-set-pos (s, *pos*), c)) = 'run)
$\wedge$ (*pos* = dv (s-pos (s), 1))
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ s-good-statep (s, c)
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ lr-check-resourcesp (*flag*, s, l, c)
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ (p-psw (lr-funcall (s->lr1 (s, l, *table*),
lr-eval ('list, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c)))
= 'run)

THEOREM: lessp-max-arg2
max (x, y) $\not<$ y

THEOREM: not-lessp-plus-arity-length-formals-alt
(listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ ($\neg$ subrp (car (s-expr (s))))
$\wedge$ litatom (car (s-expr (s)))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ s-good-statep (s, c))
$\rightarrow$ (((x + arity (car (s-expr (s)))) < length (formals (car (s-expr (s)))))
= **f**)

THEOREM: listp-lr-compile-programs
listp (lr-compile-programs (*progs*, *table*)) = listp (*progs*)

THEOREM: caar-lr-compile-programs
listp (*progs*) $\rightarrow$ (caar (lr-compile-programs (*progs*, *table*)) = caar (*progs*))

THEOREM: length-p-temp-stk-lr-eval-lr-funcall
let *lr-eval* **be** lr-eval ('list, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c)
in
(listp (s-expr (s))
$\wedge$ ($\neg$ subrp (car (s-expr (s)))))

$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ litatom (car (s-expr (s)))
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ s-good-statep (s, c)
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ (p-psw (*lr-eval*) = 'run)
$\wedge$ (p-psw (lr-funcall (s->lr1 (s, l, *table*), *lr-eval*)) = 'run)
$\wedge$ (s-err-flag (s-eval ('list, s-set-pos (s, *pos*), c)) = 'run)
$\wedge$ (*pos* = dv (s-pos (s), 1)))
$\rightarrow$ (length (p-temp-stk (lr-funcall (s->lr1 (s, l, *table*), *lr-eval*)))
= length (p-temp-stk (l))) **endlet**

THEOREM: p-ctrl-stk-size-p-ctrl-stk-lr-funcall
(p-psw (lr-funcall (l, *new-l*)) = 'run)
$\rightarrow$ (p-ctrl-stk-size (p-ctrl-stk (lr-funcall (l, *new-l*))))
= (2
+ length (formal-vars (assoc (user-fname (car (lr-expr (l))),
p-prog-segment (l))))
+ length (temp-var-dcls (assoc (user-fname (car (lr-expr (l))),
p-prog-segment (l))))
+ p-ctrl-stk-size (p-ctrl-stk (*new-l*))))

THEOREM: lr-programs-properp-s->lr1-definedp-s-progs
(lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ ($\neg$ subrp (car (s-expr (s))))
$\wedge$ litatom (car (s-expr (s)))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ s-programs-okp (cdr (s-progs (s))))
$\rightarrow$ definedp (user-fname (car (s-expr (s))), s-progs (s))

THEOREM: s-eval-ctrl-heap-temp-ws-s-fun-call-state-opener
let *s-eval* **be** s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)
in
(($c \not\simeq$ 0)
$\wedge$ listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ ($\neg$ subrp (car (s-expr (s))))
$\wedge$ litatom (car (s-expr (s)))
$\wedge$ s-good-statep (s, c)

$\wedge$ (s-err-flag (*s-eval*) = 'run)
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ ((s-eval-ctrl-r (*flag*, *s*, *c*)
= max (s-eval-ctrl-r ('list,
s-set-pos (*s*, dv (s-pos (*s*), 1)),
c),
2
+ length (s-formals (assoc (user-fname (car (s-expr (*s*))),
s-progs (*s*))))
+ length (s-temp-list (assoc (user-fname (car (s-expr (*s*))),
s-progs (*s*))))
+ s-eval-ctrl-r (**t**,
s-fun-call-state (*s-eval*,
car (s-expr (*s*))),
c − 1)))
$\wedge$ (s-eval-heap-r (*flag*, *s*, *c*)
= (s-eval-heap-r ('list,
s-set-pos (*s*, dv (s-pos (*s*), 1)),
c)
+ s-eval-heap-r (**t**,
s-fun-call-state (*s-eval*,
car (s-expr (*s*))),
c − 1)))
$\wedge$ (s-eval-temp-r (*flag*, *s*, *c*)
= max (s-eval-temp-r ('list,
s-set-pos (*s*, dv (s-pos (*s*), 1)),
c),
s-eval-temp-r (**t**,
s-fun-call-state (*s-eval*,
car (s-expr (*s*))),
c − 1)))
$\wedge$ (s-eval-ws-r (*flag*, *s*, *c*)
= max (s-eval-ws-r ('list,
s-set-pos (*s*, dv (s-pos (*s*), 1)),
c),
s-eval-ws-r (**t**,
s-fun-call-state (*s-eval*,
car (s-expr (*s*))),
c − 1)))) **endlet**

THEOREM: lr-check-resourcesp-lr-funcall-s-fun-call-state
((*c* $\not\simeq$ 0)
$\wedge$ listp (s-expr (*s*))
$\wedge$ (car (s-expr (*s*)) $\neq$ 'if)

$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-TEST)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-FETCH)
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ ($\neg$ subrp (car (s-expr (s))))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ litatom (car (s-expr (s)))
$\wedge$ lr-programs-properp (s->lr1 (s, l, *table*), *table*)
$\wedge$ (p-psw (lr-eval ('list, lr-set-pos (s->lr1 (s, l, *table*), *pos*), c))
= 'run)
$\wedge$ (s-err-flag (s-eval ('list, s-set-pos (s, *pos*), c)) = 'run)
$\wedge$ (pos = dv (s-pos (s), 1))
$\wedge$ proper-p-statep (lr->p (s->lr1 (s, l, *table*)))
$\wedge$ s-good-statep (s, c)
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ lr-check-resourcesp (*flag*, s, l, c)
$\wedge$ (*flag* $\neq$ 'list)
$\wedge$ lr-s-similar-statesp (s-params (s), s-temps (s), s->lr1 (s, l, *table*), *table*))
$\rightarrow$ lr-check-resourcesp (**t**,
s-fun-call-state (s-eval ('list,
s-set-pos (s, *pos*),
c),
car (s-expr (s))),
lr-funcall (s->lr1 (s, l, *table*),
lr-eval ('list,
lr-set-pos (s->lr1 (s, l, *table*),
pos),
c)),
c $-$ 1)

EVENT: Disable s-eval-ctrl-heap-temp-ws-s-fun-call-state-opener.

THEOREM: s-eval-flag-run-car-s-apply-subr-r-not-zero
(listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ (s-err-flag (s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)) = 'run)
$\wedge$ subrp (car (s-expr (s)))
$\wedge$ (p-psw (lr-apply-subr (s->lr1 (s, l, *table*), *new-l*)) = 'run)
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ s-good-statep (s, c))
$\rightarrow$ (car (s-apply-subr-r (car (s-expr (s)),
s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)))
$\not<$ 1)

THEOREM: length-p-temp-stk-lr-eval-lr-set-pos-flag-t
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ good-posp1 (*pos*, s-body (s-prog (*s*)))
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ s-good-statep (*s*, *c*)
 ∧ (p-psw (lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)) = 'run))
→ (length (p-temp-stk (lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), *pos*), *c*)))
 = (1 + length (p-temp-stk (*l*))))

THEOREM: s-eval-flag-run-s-eval-temp-r-not-zero
((p-psw (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)) = 'run)
 ∧ (s-err-flag (s-eval (*flag*, *s*, *c*)) = 'run)
 ∧ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
 ∧ s-good-statep (*s*, *c*)
 ∧ (*flag* ≠ 'list))
→ (s-eval-temp-r (*flag*, *s*, *c*) ≮ 1)

THEOREM: p-psw-run-p-psw-lr-if-ok-not-run-check-resourcesp
((*flag* ≠ 'list)
 ∧ (*c* ≯ 0)
 ∧ listp (s-expr (*s*))
 ∧ (car (s-expr (*s*)) = 'if)
 ∧ s-good-statep (*s*, *c*)
 ∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
 ∧ proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
 ∧ (s-err-flag (s-eval (**t**, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*)) = 'run)
 ∧ (p-psw (lr-if-ok (lr-eval (**t**,
 lr-set-pos (s->lr1 (*s*, *l*, *table*), dv (s-pos (*s*), 1)),
 c)))
 ≠ 'run)
 ∧ (p-psw (lr-eval (**t**, lr-set-pos (s->lr1 (*s*, *l*, *table*), dv (s-pos (*s*), 1)), *c*))
 = 'run)
 ∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*))))
→ (¬ lr-check-resourcesp (*flag*, *s*, *l*, *c*))

THEOREM: not-lr-check-resourcesp-temp-test-bad-max-temp-stk-size
((*flag* ≠ 'list)
 ∧ (*c* ≠ 0)
 ∧ (*c* ∈ **N**)
 ∧ listp (s-expr (*s*))
 ∧ (car (s-expr (*s*)) = S-TEMP-TEST)
 ∧ (p-max-temp-stk-size (*l*) < (2 + length (p-temp-stk (*l*))))
 ∧ s-good-statep (*s*, *c*))
→ (¬ lr-check-resourcesp (*flag*, *s*, *l*, *c*))

THEOREM: lr-eval-s-eval-flag-run
(proper-p-statep (lr->p (s->lr1 (*s*, *l*, *table*)))
∧ lr-proper-heapp (p-data-segment (*l*))
∧ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
∧ lr-programs-properp (s->lr1 (*s*, *l*, *table*), *table*)
∧ lr-s-similar-statesp (s-params (*s*), s-temps (*s*), s->lr1 (*s*, *l*, *table*), *table*)
∧ s-good-statep (*s*, *c*)
∧ s-all-temps-setp (*flag*,
if *flag* = 'list **then** s-expr-list (*s*)
else s-expr (*s*) **endif**,
temp-alist-to-set (s-temps (*s*)))
∧ s-all-progs-temps-setp (s-progs (*s*))
∧ s-check-temps-setp (s-temps (*s*))
∧ (s-err-flag (s-eval (*flag*, *s*, *c*)) = 'run)
∧ lr-check-resourcesp (*flag*, *s*, *l*, *c*)
∧ (p-word-size (*l*) ≮ S-MAX-SUBR-REQS))
→ (p-psw (lr-eval (*flag*, s->lr1 (*s*, *l*, *table*), *c*)) = 'run)

THEOREM: plistp-lr-compile-body-1
plistp (*prog*) → plistp (lr-compile-body (*flag*, *prog*, *temp-alist*, *table*))

DEFINITION:
l-restrict-subrps (*flag*, *expr*)
= **if** *flag* = 'list
then if listp (*expr*)
then l-restrict-subrps (**t**, car (*expr*))
∧ l-restrict-subrps ('list, cdr (*expr*))
else t endif
elseif listp (*expr*)
then if car (*expr*) = 'quote **then t**
elseif car (*expr*) = 'if
then l-restrict-subrps ('list, cdr (*expr*))
elseif subrp (car (*expr*))
then definedp (car (*expr*), P-RUNTIME-SUPPORT-PROGRAMS)
∧ l-restrict-subrps ('list, cdr (*expr*))
elseif body (car (*expr*)) **then** l-restrict-subrps ('list, cdr (*expr*))
else t endif
else t endif

DEFINITION:
l-restrict-subrps-progs (*pnames*)
= **if** listp (*pnames*)
then l-restrict-subrps (**t**, body (car (*pnames*)))
∧ l-restrict-subrps-progs (cdr (*pnames*))
else t endif

DEFINITION:
s-restrict-subrps (*flag*, *expr*)
= **if** *flag* = 'list
then if listp (*expr*)
then s-restrict-subrps (**t**, car (*expr*))
∧ s-restrict-subrps ('list, cdr (*expr*))
else t endif
elseif listp (*expr*)
then if car (*expr*) = 'quote **then t**
elseif (car (*expr*) = S-TEMP-FETCH)
∨ (car (*expr*) = S-TEMP-EVAL)
∨ (car (*expr*) = S-TEMP-TEST)
then s-restrict-subrps (**t**, cadr (*expr*))
elseif car (*expr*) = 'if
then s-restrict-subrps ('list, cdr (*expr*))
elseif subrp (car (*expr*))
then definedp (car (*expr*), P-RUNTIME-SUPPORT-PROGRAMS)
∧ s-restrict-subrps ('list, cdr (*expr*))
elseif body (car (*expr*)) **then** s-restrict-subrps ('list, cdr (*expr*))
else t endif
else t endif

DEFINITION:
s-restrict-subrps-progs (*progs*)
= **if** listp (*progs*)
then s-restrict-subrps (**t**, s-body (car (*progs*)))
∧ s-restrict-subrps-progs (cdr (*progs*))
else t endif

THEOREM: s-proper-exprp-plist-temp-list
s-proper-exprp (*flag*, *expr*, *program-names*, *formals*, plist (*temp-list*))
= s-proper-exprp (*flag*, *expr*, *program-names*, *formals*, *temp-list*)

THEOREM: not-listp-s-progs-not-s-good-statep
(¬ listp (s-progs (*s*))) → (¬ s-good-statep (*s*, *c*))

THEOREM: length-lr-init-heap-contents
length (lr-init-heap-contents (*addr*, *size*)) = (1 + (*size* ∗ LR-NODE-SIZE))

THEOREM: fetch-cons
fetch (list (*x*, cons (*name1*, *n*)), cons (cons (*name2*, *contents*), *rest-data-seg*))
= **if** *name1* = *name2* **then** get (*n*, *contents*)
else fetch (list (*x*, cons (*name1*, *n*)), *rest-data-seg*) **endif**

THEOREM: lr-s-similar-const-table-cons

lr-s-similar-const-table (cons (cons (*object*, *addr*), *table*), *data-seg*)
= (lr-valp (*object*, *addr*, *data-seg*)
∧ lr-s-similar-const-table (*table*, *data-seg*))

THEOREM: lr-s-similar-const-table-nil
lr-s-similar-const-table (**nil**, *data-seg*)

THEOREM: lr-init-heap-contents-add1-opener
lr-init-heap-contents (*addr*, 1 + *size*)
= append (lr-new-node (tag ('nat, LR-INIT-TAG),
add-addr (*addr*, LR-NODE-SIZE),
tag ('nat, 0),
tag ('nat, 0)),
lr-init-heap-contents (add-addr (*addr*, LR-NODE-SIZE), *size*))

THEOREM: deposit-cons
deposit (*object*,
list (*x*, cons (*name1*, *n*)),
cons (cons (*name2*, *contents*), *rest-data-seg*))
= **if** *name1* = *name2*
then cons (cons (*name1*, put (*object*, *n*, *contents*)), *rest-data-seg*)
else cons (cons (*name2*, *contents*),
deposit (*object*, list (*x*, cons (*name1*, *n*)), *rest-data-seg*)) **endif**

THEOREM: adpp-cons-pack-opener
(*n* ∈ **N**)
→ (adpp (cons (pack (*xxx*), *n*), cons (cons (pack (*yyy*), *contents*), *rest*))
= **if** *xxx* = *yyy* **then** *n* < length (*contents*)
else adpp (cons (pack (*xxx*), *n*), *rest*) **endif**)

THEOREM: fetch-deposit-a-list
((offset (*addr1*) ∈ **N**) ∧ (offset (*addr2*) ∈ **N**) ∧ listp (*list*))
→ (fetch (*addr1*, deposit-a-list (*list*, *addr2*, *data-seg*))
= **if** definedp (area-name (*addr2*), *data-seg*)
then if area-name (*addr1*) = area-name (*addr2*)
then if (offset (*addr1*) ≮ offset (*addr2*))
∧ (offset (*addr1*)
< (offset (*addr2*) + length (*list*)))
then get (offset (*addr1*) − offset (*addr2*), *list*)
else fetch (*addr1*, *data-seg*) **endif**
else fetch (*addr1*, *data-seg*) **endif**
else fetch (*addr1*, *data-seg*) **endif**)

THEOREM: lr-valp-0-lr-0-addr-opener
lr-valp (0, identity (LR-0-ADDR), *data-seg*)

```
=   (adpp (identity (untag (LR-0-ADDR)), data-seg)
    ∧   (type (fetch (identity (add-addr (LR-0-ADDR, LR-REF-COUNT-OFFSET)),
                     data-seg))
            =   'nat)
    ∧   (untag (fetch (identity (LR-0-ADDR), data-seg)) = LR-ADD1-TAG)
    ∧   (untag (fetch (identity (add-addr (LR-0-ADDR, LR-UNBOX-NAT-OFFSET)),
                      data-seg))
            =   0))
```

EVENT: Disable lr-valp-0-lr-0-addr-opener.

THEOREM: lr-valp-t-lr-t-addr-opener

```
lr-valp (t, identity (LR-T-ADDR), data-seg)
=   (adpp (identity (untag (LR-T-ADDR)), data-seg)
    ∧   (type (fetch (identity (add-addr (LR-T-ADDR, LR-REF-COUNT-OFFSET)),
                     data-seg))
            =   'nat)
    ∧   (untag (fetch (identity (LR-T-ADDR), data-seg)) = LR-TRUE-TAG))
```

EVENT: Disable lr-valp-t-lr-t-addr-opener.

THEOREM: lr-valp-f-lr-f-addr-opener

```
lr-valp (f, identity (LR-F-ADDR), data-seg)
=   (adpp (identity (untag (LR-F-ADDR)), data-seg)
    ∧   (type (fetch (identity (add-addr (LR-F-ADDR, LR-REF-COUNT-OFFSET)),
                     data-seg))
            =   'nat)
    ∧   (untag (fetch (identity (LR-F-ADDR), data-seg)) = LR-FALSE-TAG))
```

EVENT: Disable lr-valp-f-lr-f-addr-opener.

THEOREM: definedp-table-definedp-cdr-lr-compile-quote

```
definedp (x, table)
→   definedp (x, cdr (lr-compile-quote (flag, object, data-seg, table)))
```

THEOREM: definedp-car-lr-compile-quote

```
definedp (x, car (lr-compile-quote (flag, object, data-seg, table)))
=   definedp (x, data-seg)
```

THEOREM: lr-proper-p-areasp-car-lr-compile-quote

```
lr-proper-p-areasp (data-seg)
→   lr-proper-p-areasp (car (lr-compile-quote (flag, object, data-seg, table)))
```

THEOREM: length-deposit-a-list
listp (*list*)
→ (length (cdr (assoc (*name*, deposit-a-list (*list*, *addr*, *data-seg*))))
= **if** definedp (area-name (*addr*), *data-seg*)
then if area-name (*addr*) = *name*
then if length (cdr (assoc (*name*, *data-seg*)))
< (offset (*addr*) + length (*list*))
then offset (*addr*) + length (*list*)
else length (cdr (assoc (*name*, *data-seg*))) **endif**
else length (cdr (assoc (*name*, *data-seg*))) **endif**
else length (cdr (assoc (*name*, *data-seg*))) **endif**)

THEOREM: adpp-lr-compile-quote
adpp (*addr*, *data-seg*)
→ adpp (*addr*, car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))

THEOREM: adpp-untag-definedp-area-name-free-ptr
adpp (untag (LR-FP-ADDR), *data-seg*)
→ definedp (identity (area-name (LR-FP-ADDR)), *data-seg*)

THEOREM: lr-max-node-deposit-a-list
(adpp (untag (*addr*), *data-seg*)
∧ listp (*list*)
∧ ((offset (*addr*) + length (*list*))
< length (cdr (assoc (area-name (*addr*), *data-seg*)))))
→ (lr-max-node (deposit-a-list (*list*, *addr*, *data-seg*))
= lr-max-node (*data-seg*))

DEFINITION:
all-p-objects-lookup (*list*, *table*, *p*)
= **if** listp (*list*)
then p-objectp (cdr (assoc (car (*list*), *table*)), *p*)
∧ all-p-objects-lookup (cdr (*list*), *table*, *p*)
else t endif

THEOREM: proper-p-alistp-all-litatoms-all-p-objectps-lookup
(all-litatoms (strip-cars (*params*))
∧ all-p-objects-lookup (strip-cdrs (*params*), *table*, *p*))
→ proper-p-alistp (pair-formals-with-addresses (*params*, *table*), *p*)

THEOREM: definedp-table-definedp-cdr-lr-data-seg-table-body
definedp (*object*, *table*)
→ definedp (*object*, cdr (lr-data-seg-table-body (*flag*, *expr*, *data-seg*, *table*)))

THEOREM: definedp-table-definedp-cdr-lr-data-seg-table-list
definedp (*object*, *table*)
→ definedp (*object*, cdr (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))

THEOREM: definedp-table-definedp-cdr-lr-init-data-seg-table
definedp (*object*, *table*)
→ definedp (*object*, cdr (lr-init-data-seg-table (*params*, *data-seg*, *table*)))

THEOREM: definedp-table-definedp-car-lr-data-seg-table-body
definedp (*name*, *data-seg*)
→ definedp (*name*, car (lr-data-seg-table-body (*flag*, *expr*, *data-seg*, *table*)))

THEOREM: definedp-table-definedp-car-lr-data-seg-table-list
definedp (*name*, *data-seg*)
→ definedp (*name*, car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))

THEOREM: equal-lengths-same-signature-car-lr-compile-quote
same-signature (*data-seg*, car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))
→ (length (cdr (assoc (*name*,
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))))
= length (cdr (assoc (*name*, *data-seg*))))

THEOREM: adpp-same-signature-car-lr-compile-quote
same-signature (*data-seg*, car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))
→ (adpp (*adp*, car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))
= adpp (*adp*, *data-seg*))

THEOREM: same-signature-car-lr-compile-quote-helper
let *pair* **be** lr-compile-quote (`'list`,
list (car (*object*), cdr (*object*)),
data-seg,
table)
in
(lr-proper-free-listp (car (*pair*))
∧ same-signature (*data-seg*, car (*pair*))
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*))
∧ ((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) − 1)
$\not<$ (offset (fetch (LR-FP-ADDR, car (*pair*))) + length (*list*))))
→ same-signature (*data-seg*,
deposit-a-list (*list*,
fetch (identity (LR-FP-ADDR),
car (*pair*)),
car (*pair*))) **endlet**

THEOREM: same-signature-car-lr-compile-quote-generalized
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)

```
 ∧   definedp (LR-HEAP-NAME, data-seg)
 ∧   lr-boundary-nodep (lr-max-node (data-seg)))
 →   (same-signature (data-seg,
                      car (lr-compile-quote (flag, object, data-seg, table)))
      ∧   lr-proper-free-listp (car (lr-compile-quote (flag,
                                                       object,
                                                       data-seg,
                                                       table))))
```

EVENT: Disable equal-lengths-same-signature-car-lr-compile-quote.

EVENT: Disable adpp-same-signature-car-lr-compile-quote.

THEOREM: lr-proper-free-listp-car-lr-compile-quote

```
 (lr-proper-free-listp (data-seg)
  ∧   lr-proper-p-areasp (data-seg)
  ∧   definedp (LR-HEAP-NAME, data-seg)
  ∧   lr-boundary-nodep (lr-max-node (data-seg)))
 →   lr-proper-free-listp (car (lr-compile-quote (flag, object, data-seg, table)))
```

THEOREM: p-objectp-car-lr-compile-quote

```
 (p-objectp (object1,
             p-state (pc,
                      ctrl-stk,
                      temp-stk,
                      prog-seg,
                      data-seg,
                      max-ctrl,
                      max-temp,
                      word-size,
                      psw))
  ∧   lr-proper-free-listp (data-seg)
  ∧   lr-proper-p-areasp (data-seg)
  ∧   definedp (LR-HEAP-NAME, data-seg)
  ∧   lr-boundary-nodep (lr-max-node (data-seg)))
 →   p-objectp (object1,
                p-state (pc,
                         ctrl-stk,
                         temp-stk,
                         prog-seg,
                         car (lr-compile-quote (flag, object2, data-seg, table)),
                         max-ctrl,
                         max-temp,
```

word-size,
psw))

THEOREM: lr-proper-p-areasp-car-lr-data-seg-table-body
lr-proper-p-areasp (*data-seg*)
→ lr-proper-p-areasp (car (lr-data-seg-table-body (*flag*,
expr,
data-seg,
table)))

THEOREM: same-signature-car-lr-compile-quote
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ same-signature (*data-seg*,
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))

THEOREM: same-signature-car-lr-data-seg-table-body
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (same-signature (*data-seg*,
car (lr-data-seg-table-body (*flag*, *expr*, *data-seg*, *table*)))
∧ lr-proper-free-listp (car (lr-data-seg-table-body (*flag*,
expr,
data-seg,
table))))

EVENT: Disable same-signature-car-lr-compile-quote.

THEOREM: lr-max-node-car-lr-data-seg-table-body
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (lr-max-node (car (lr-data-seg-table-body (*flag*, *body*, *data-seg*, *table*)))
= lr-max-node (*data-seg*))

THEOREM: same-signature-car-lr-data-seg-table-list-helper
let *dst-body* **be** lr-data-seg-table-body (**t**, s-body (*prog*), *data-seg*, *table*)
in
(same-signature (car (*dst-body*),

car (lr-data-seg-table-list (*progs*,
car (*dst-body*),
cdr (*dst-body*))))
∧ lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ same-signature (*data-seg*,
car (lr-data-seg-table-list (*progs*,
car (*dst-body*),
cdr (*dst-body*)))) **endlet**

THEOREM: same-signature-car-lr-data-seg-table-list
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ same-signature (*data-seg*,
car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))

EVENT: Disable same-signature-car-lr-data-seg-table-list-helper.

THEOREM: length-car-lr-compile-quote
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (length (cdr (assoc (*name*,
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*))))))
= length (cdr (assoc (*name*, *data-seg*))))

THEOREM: lr-max-node-car-lr-compile-quote
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (lr-max-node (car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))
= lr-max-node (*data-seg*))

THEOREM: lr-proper-free-listp-car-lr-init-data-seg-table
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ lr-proper-free-listp (car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))

THEOREM: adpp-untag-lr-fp-addr-lr-init-data-seg
adpp (identity (untag (LR-FP-ADDR)), lr-init-data-seg (*heap-size*))

THEOREM: lr-max-node-lr-init-data-seg
(*heap-size* $\not<$ 2)
→ (lr-max-node (lr-init-data-seg (*heap-size*))
= tag ('addr,
cons (identity (LR-HEAP-NAME),
identity (LR-NODE-SIZE) ∗ *heap-size*)))

THEOREM: fetch-lr-fp-addr-lr-init-data-seg
fetch (identity (LR-FP-ADDR), lr-init-data-seg (*heap-size*))
= identity (add-addr (LR-F-ADDR, LR-NODE-SIZE))

THEOREM: lr-boundary-nodep-not-lessp-fact-helper
((*x* < (*y* ∗ *z*)) ∧ ((*x* **mod** *y*) = 0) ∧ (*x* ∈ **N**))
→ ((*x* < (*y* ∗ (*z* − 1))) = (*x* ≠ (*y* ∗ (*z* − 1))))

THEOREM: lessp-times-difference-fact
((*z* ≠ 0) ∧ (*x* ≠ 0) ∧ ((*x* **mod** *y*) = 0))
→ (((*x* − *y*) < (*y* ∗ (*z* − 1))) = (*x* < (*y* ∗ *z*)))

THEOREM: lessp-times-difference-node-on-boundaryp-fact
((*heap-size* ≠ 0) ∧ (offset (*addr*) ≠ 0) ∧ lr-boundary-nodep (*addr*))
→ ((((((offset (*addr*) − 1) − 1) − 1) − 1)
< (identity (LR-NODE-SIZE) ∗ (*heap-size* − 1)))
= (offset (*addr*) < (identity (LR-NODE-SIZE) ∗ *heap-size*)))

THEOREM: lr-boundary-nodep-lessp-lr-node-size-0
lr-boundary-nodep (*addr*)
→ ((((offset (*addr*) − 1) = 1) = **f**)
∧ ((((offset (*addr*) − 1) − 1) = 1) = **f**))

THEOREM: lr-boundary-nodep-lessp-lr-node-size-1
((offset (*addr*) ∈ **N**) ∧ lr-boundary-nodep (*addr*) ∧ (*n* < LR-NODE-SIZE))
→ ((*n* < offset (*addr*)) = (offset (*addr*) ≠ 0))

THEOREM: lr-boundary-nodep-lessp-lr-node-size-2
lr-boundary-nodep (*addr*) → ((offset (*addr*) = 1) = **f**)

DEFINITION:
induct-hint-17 (*addr1*, *size*, *addr2*)
= **if** *size* ≃ 0 **then t**
elseif offset (*addr2*) ≃ 0 **then t**
else induct-hint-17 (add-addr (*addr1*, LR-NODE-SIZE),
size − 1,
sub-addr (*addr2*, LR-NODE-SIZE)) **endif**

THEOREM: get-cdr-lr-init-heap-contents
((offset (*addr2*) < (LR-NODE-SIZE ∗ *heap-size*))
∧ lr-boundary-nodep (*addr2*)
∧ (offset (*addr2*) ∈ **N**)
∧ (offset (*addr1*) ∈ **N**))
→ (get (offset (*addr2*), cdr (lr-init-heap-contents (*addr1*, *heap-size*)))
= add-addr (add-addr (*addr1*, offset (*addr2*)), LR-NODE-SIZE))

EVENT: Disable lr-boundary-nodep-lessp-lr-node-size-0.

EVENT: Disable lr-boundary-nodep-lessp-lr-node-size-1.

EVENT: Disable lr-boundary-nodep-lessp-lr-node-size-2.

THEOREM: length-cdr-assoc-lr-heap-name-lr-init-data-seg
(*heap-size* ≮ 2)
→ (length (cdr (assoc (identity (LR-HEAP-NAME), lr-init-data-seg (*heap-size*))))
= (1 + (*heap-size* ∗ identity (LR-NODE-SIZE))))

THEOREM: fetch-add-addr-ref-count-offset-lr-init-data-seg-help-1
((offset (*addr*) = 0)
∧ (type (*addr*) = 'addr)
∧ (area-name (*addr*) = 'heap))
→ (add-addr (*addr*, 4) = '(addr (heap . 4)))

THEOREM: equal-add-addr-fact
(type (*addr1*) = type (*addr2*))
→ ((add-addr (*addr1*, *n1*) = add-addr (*addr2*, *n2*))
= ((area-name (*addr1*) = area-name (*addr2*))
∧ ((offset (*addr1*) + *n1*) = (offset (*addr2*) + *n2*))))

DEFINITION:
lr-all-nodes (*min-offset*, *max-addr*)
= **if** offset (*max-addr*) ≃ 0 **then nil**
elseif *min-offset* ≮ offset (*max-addr*) **then nil**
else cons (sub-addr (*max-addr*, LR-NODE-SIZE),
lr-all-nodes (*min-offset*,
sub-addr (*max-addr*, LR-NODE-SIZE))) **endif**

DEFINITION:
induct-hint-19 (*addr*, *max-addr*)
= **if** offset (*addr*) < offset (*max-addr*)
then induct-hint-19 (add-addr (*addr*, LR-NODE-SIZE), *max-addr*)
else t endif

THEOREM: lessp-times-plus-fact
$(n \not\simeq \mathtt{0}) \rightarrow (((n * v) < (n + (n * w))) = (v < (1 + w)))$

THEOREM: lessp-sub1-lessp-fact
$((x \in \mathbf{N}) \wedge (y \in \mathbf{N}) \wedge (x \neq \mathtt{0}) \wedge (x \neq y))$
$\rightarrow \quad (((x - 1) < y) = (x < y))$

THEOREM: remainder-difference-not-equal-lessp-fact
$(((x \textbf{ mod } n) = \mathtt{0})$
$\wedge \quad ((y \textbf{ mod } n) = \mathtt{0})$
$\wedge \quad (x \neq (y - n))$
$\wedge \quad (y \not< n)$
$\wedge \quad (x \in \mathbf{N})$
$\wedge \quad (y \in \mathbf{N}))$
$\rightarrow \quad ((x < (y - n)) = (x < y))$

EVENT: Disable lessp-sub1-lessp-fact.

THEOREM: lr-boundaryp-nodep-difference-node-size
lr-boundary-offsetp (*offset*)
$\rightarrow$ lr-boundary-offsetp (*offset* $-$ LR-NODE-SIZE)

THEOREM: lr-boundary-offsetp-difference-not-equal-lessp-fact-1
(lr-boundary-offsetp (x)
$\wedge$ lr-boundary-offsetp (y)
$\wedge \quad (x \neq (y -$ LR-NODE-SIZE))
$\wedge \quad (y \not<$ LR-NODE-SIZE)
$\wedge \quad (x \in \mathbf{N})$
$\wedge \quad (y \in \mathbf{N}))$
$\rightarrow \quad ((x < (y -$ LR-NODE-SIZE)) $= (x < y))$

THEOREM: member-lr-all-nodes-helper
((offset (*max-addr*) $\not\simeq \mathtt{0}$)
$\wedge$ (offset (*addr*) $\in \mathbf{N}$)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ listp (untag (*addr*))
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ lr-boundary-nodep (*max-addr*)
$\wedge$ (area-name (*addr*) = area-name (*max-addr*))
$\wedge$ (type (*addr*) = type (*max-addr*))
$\wedge \quad (addr \neq$ sub-addr (*max-addr*, LR-NODE-SIZE)))
$\rightarrow \quad ((\text{offset}\,(addr) < ((((\text{offset}\,(max\text{-}addr) - 1) - 1) - 1) - 1))$
$= \quad (\text{offset}\,(addr) < \text{offset}\,(max\text{-}addr)))$

THEOREM: member-lr-all-nodes
((type (*addr*) = 'addr)
 ∧ (cddr (*addr*) = **nil**)
 ∧ listp (*addr*)
 ∧ lr-boundary-nodep (*addr*)
 ∧ (area-name (*addr*) = LR-HEAP-NAME)
 ∧ listp (untag (*addr*))
 ∧ (offset (*addr*) ∈ **N**)
 ∧ (type (*max-addr*) = 'addr)
 ∧ (cddr (*max-addr*) = **nil**)
 ∧ listp (*max-addr*)
 ∧ lr-boundary-nodep (*max-addr*)
 ∧ (area-name (*max-addr*) = LR-HEAP-NAME)
 ∧ listp (untag (*max-addr*))
 ∧ lr-boundary-offsetp (*min-offset*)
 ∧ (offset (*addr*) ≮ *min-offset*))
→ ((*addr* ∈ lr-all-nodes (*min-offset*, *max-addr*))
  = (offset (*addr*) < offset (*max-addr*))))

EVENT: Disable member-lr-all-nodes-helper.

THEOREM: lr-all-nodes-nil
(lr-all-nodes (*min-offset*, *max-addr*) = **nil**)
= ((offset (*max-addr*) ≃ 0) ∨ (*min-offset* ≮ offset (*max-addr*)))

THEOREM: delete-append
delete (*e*, append (*x*, *y*))
= **if** *e* ∈ *x* **then** append (delete (*e*, *x*), *y*)
  else append (*x*, delete (*e*, *y*)) **endif**

THEOREM: lessp-difference-node-size-sub-addr-2
((*offset* < offset (*addr*))
 ∧ lr-boundary-nodep (*addr*)
 ∧ (offset (*addr*) ∈ **N**)
 ∧ lr-boundary-offsetp (*offset*))
→ (((offset (*addr*) − identity (LR-NODE-SIZE)) < *offset*) = **f**)

THEOREM: not-member-lr-all-nodes-too-small-addr
(lr-boundary-nodep (*addr*)
 ∧ lr-boundary-nodep (*max-addr*)
 ∧ lr-boundary-offsetp (*min-offset*)
 ∧ (offset (*addr*) < *min-offset*)
 ∧ (*min-offset* ∈ **N**))
→ (*addr* ∉ lr-all-nodes (*min-offset*, *max-addr*))

THEOREM: plist-delete
plist (delete (*e*, *x*)) = delete (*e*, plist (*x*))

THEOREM: lr-check-free-nodes-plist-node-list
lr-check-free-nodes (*addr*, plist (*node-list*), *data-seg*, *max-addr*)
= lr-check-free-nodes (*addr*, *node-list*, *data-seg*, *max-addr*)

THEOREM: lr-all-nodes-offset-same-max
lr-all-nodes (offset (*addr*), *addr*) = **nil**

THEOREM: lr-all-nodes-offset-max-addr-opener-helper
((offset (*addr*) ≠ 0)
∧ lr-boundary-nodep (*addr*)
∧ (*offset* ∈ **N**)
∧ lr-boundary-offsetp (*offset*)
∧ (*offset* < offset (*addr*)))
→ ((*offset* < ((((offset (*addr*) − 1) − 1) − 1) − 1))
= (*offset* ≠ ((((offset (*addr*) − 1) − 1) − 1) − 1)))

THEOREM: lr-all-nodes-lessp-max-addr-opener
((type (*max-addr*) = 'addr)
∧ listp (*max-addr*)
∧ (cddr (*max-addr*) = **nil**)
∧ listp (untag (*max-addr*))
∧ (offset (*max-addr*) ∈ **N**)
∧ lr-boundary-nodep (*max-addr*)
∧ (area-name (*max-addr*) = LR-HEAP-NAME)
∧ (*min-offset* < offset (*max-addr*))
∧ (*min-offset* ∈ **N**)
∧ lr-boundary-offsetp (*min-offset*))
→ (lr-all-nodes (*min-offset*, *max-addr*)
= append (lr-all-nodes (*min-offset* + identity (LR-NODE-SIZE),
max-addr),
list (tag ('addr,
cons (identity (LR-HEAP-NAME), *min-offset*)))))

THEOREM: fetch-init-init-data-seg-generalized
((offset (*addr*) ∈ **N**)
∧ (type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ listp (untag (*addr*))
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = LR-HEAP-NAME)
∧ (offset (*addr*) < (identity (LR-NODE-SIZE) ∗ *heap-size*))

$\wedge$ (cdr (assoc (LR-HEAP-NAME, *data-seg*))
= lr-init-heap-contents (identity (tag (`'addr`,
cons (LR-HEAP-NAME, `0`))),
heap-size)))
$\rightarrow$ (fetch (add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)), *data-seg*)
= add-addr (*addr*, `4`))

THEOREM: lessp-difference-node-size-sub-addr-3
((offset (*addr*) < (LR-NODE-SIZE $*$ *heap-size*))
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ (offset (*addr*) $\in$ **N**))
$\rightarrow$ (((((((identity (LR-NODE-SIZE) $*$ *heap-size*) $-$ 1) $-$ 1) $-$ 1) $-$ 1)
< offset (*addr*))
= **f**)

THEOREM: lr-boundary-nodep-tag-cons-times-lr-node-size
lr-boundary-nodep (tag (*x*, cons (*name*, identity (LR-NODE-SIZE) $*$ *heap-size*)))

THEOREM: tag-type-name-offset-equal-same
((type (*addr*) = *x*)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (untag (*addr*))
$\wedge$ (offset (*addr*) $\in$ **N**)
$\wedge$ (area-name (*addr*) = *name*))
$\rightarrow$ (tag (*x*, cons (*name*, offset (*addr*))) = *addr*)

THEOREM: lr-check-free-nodes-lr-free-list-nodes-init-data-seg
let *init-data-seg* **be** list (cons (area-name (LR-FP-ADDR), *any1*),
cons (area-name (LR-ANSWER-ADDR), *any2*),
cons (LR-HEAP-NAME,
lr-init-heap-contents (tag (`'addr`,
cons (LR-HEAP-NAME,
`0`)),
heap-size)))
in
((offset (*max-addr*) $\not<$ offset (*addr*))
$\wedge$ (type (*addr*) = `'addr`)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ listp (untag (*addr*))
$\wedge$ (offset (*addr*) $\in$ **N**)
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ (area-name (*addr*) = LR-HEAP-NAME)
$\wedge$ (*max-addr* = lr-max-node (*init-data-seg*)))
$\rightarrow$ lr-check-free-nodes (*addr*,

lr-all-nodes (offset (*addr*), *max-addr*),
list (cons (identity (area-name (LR-FP-ADDR)),
any1),
cons (identity (area-name (LR-ANSWER-ADDR)),
any2),
cons (identity (LR-HEAP-NAME),
lr-init-heap-contents (identity (tag ('addr,
cons (LR-HEAP-NAME,
0))),
heap-size))),
max-addr) **endlet**

EVENT: Disable fetch-init-init-data-seg-generalized.

THEOREM: lr-free-list-nodes-deposit-a-list-lr-nodep
((type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ adpp (untag (*addr*), *data-seg*)
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = LR-HEAP-NAME)
∧ (type (*max-addr*) = 'addr)
∧ (cddr (*max-addr*) = **nil**)
∧ listp (*max-addr*)
∧ adpp (untag (*max-addr*), *data-seg*)
∧ lr-boundary-nodep (*max-addr*)
∧ (area-name (*max-addr*) = LR-HEAP-NAME))
→ (lr-free-list-nodes (*max-addr*,
deposit-a-list (list (*a*, *b*, *c*, *d*), *addr*, *data-seg*))
= lr-free-list-nodes (*max-addr*,
deposit (*b*,
add-addr (*addr*,
identity (LR-REF-COUNT-OFFSET)),
data-seg)))

THEOREM: lr-check-free-nodes-deposit-a-list-lr-nodep
((type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ adpp (untag (*addr*), *data-seg*)
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = LR-HEAP-NAME)
∧ (type (*max-addr*) = 'addr)
∧ (cddr (*max-addr*) = **nil**)

$\wedge$ listp (*max-addr*)
$\wedge$ adpp (untag (*max-addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*max-addr*)
$\wedge$ (area-name (*max-addr*) = LR-HEAP-NAME)
$\wedge$ lr-node-listp (*node-list*, *data-seg*))
$\rightarrow$ (lr-check-free-nodes (*addr1*,
node-list,
deposit-a-list (list (*a*, *b*, *c*, *d*), *addr*, *data-seg*),
max-addr)
= lr-check-free-nodes (*addr1*,
node-list,
deposit (*b*,
add-addr (*addr*,
identity (LR-REF-COUNT-OFFSET)),
data-seg),
max-addr))

THEOREM: lr-all-nodes-not-lessp-min-offset-max-addr
(*min-offset* $\not<$ offset (*max-addr*))
$\rightarrow$ (lr-all-nodes (*min-offset*, *max-addr*) = **nil**)

THEOREM: fetch-init-init-data-seg-sub-addr
(((identity (LR-NODE-SIZE) $*$ *heap-size*) $\not<$ offset (*addr*))
$\wedge$ (offset (*addr*) $\in$ **N**)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ listp (untag (*addr*))
$\wedge$ (type (*addr*) = `'addr`)
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ (area-name (*addr*) = LR-HEAP-NAME)
$\wedge$ ((offset (*addr*) $-$ identity (LR-NODE-SIZE))
$<$ (LR-NODE-SIZE $*$ *heap-size*))
$\wedge$ (offset (*addr*) $\neq$ `0`)
$\wedge$ (cdr (assoc (LR-HEAP-NAME, *data-seg*))
= lr-init-heap-contents (tag (`'addr`, cons (LR-HEAP-NAME, `0`)),
heap-size)))
$\rightarrow$ (fetch (add-addr (sub-addr (*addr*, identity (LR-NODE-SIZE)),
identity (LR-REF-COUNT-OFFSET)),
data-seg)
= *addr*)

THEOREM: lr-free-list-nodes-lr-init-heap-contents-generalized
(lr-boundary-nodep (*max-addr*)
$\wedge$ (area-name (*max-addr*) = LR-HEAP-NAME)
$\wedge$ (type (*max-addr*) = `'addr`)

$\wedge$ (cddr (*max-addr*) = **nil**)
$\wedge$ listp (*max-addr*)
$\wedge$ listp (untag (*max-addr*))
$\wedge$ ((LR-NODE-SIZE $*$ *heap-size*) $\not<$ offset (*max-addr*))
$\wedge$ (cdr (assoc (LR-HEAP-NAME, *data-seg*))
= lr-init-heap-contents (identity (tag (`'addr`,
cons (LR-HEAP-NAME, `0`))),
heap-size)))
$\rightarrow$ (lr-free-list-nodes (*max-addr*, *data-seg*) = lr-all-nodes (`0`, *max-addr*))

THEOREM: lr-free-list-nodes-lr-init-heap-contents
(lr-boundary-nodep (*max-addr*)
$\wedge$ (area-name (*max-addr*) = LR-HEAP-NAME)
$\wedge$ (type (*max-addr*) = `'addr`)
$\wedge$ (cddr (*max-addr*) = **nil**)
$\wedge$ listp (*max-addr*)
$\wedge$ listp (untag (*max-addr*))
$\wedge$ ((LR-NODE-SIZE $*$ *heap-size*) $\not<$ offset (*max-addr*)))
$\rightarrow$ (lr-free-list-nodes (*max-addr*,
list (cons (identity (area-name (LR-FP-ADDR)), *any1*),
cons (identity (area-name (LR-ANSWER-ADDR)),
any2),
cons (identity (LR-HEAP-NAME),
lr-init-heap-contents (identity (tag (`'addr`,
cons (LR-HEAP-NAME,
`0`))),
heap-size))))
= lr-all-nodes (`0`, *max-addr*))

THEOREM: lr-node-listp-lr-all-nodes
(lr-boundary-nodep (*addr*)
$\wedge$ (area-name (*addr*) = LR-HEAP-NAME)
$\wedge$ adpp (untag (*addr*), *data-seg*)
$\wedge$ (type (*addr*) = `'addr`))
$\rightarrow$ lr-node-listp (lr-all-nodes (*min-offset*, *addr*), *data-seg*)

THEOREM: plistp-lr-all-nodes
plistp (lr-all-nodes (*min-offset*, *max-addr*))

THEOREM: lr-free-list-nodes-lr-init-data-seg
(*heap-size* $\not<$ `2`)
$\rightarrow$ (lr-free-list-nodes (tag (`'addr`,
cons (identity (LR-HEAP-NAME),
identity (LR-NODE-SIZE) $*$ *heap-size*)),
lr-init-data-seg (*heap-size*))

= lr-all-nodes (identity (offset (add-addr (LR-F-ADDR, LR-NODE-SIZE))),
tag ('addr,
cons (identity (LR-HEAP-NAME),
identity (LR-NODE-SIZE) * *heap-size*))))

THEOREM: lr-proper-free-listp-lr-init-data-seg-helper
(*heap-size* $\not<$ 2)
→ lr-check-free-nodes (identity (add-addr (LR-F-ADDR, LR-NODE-SIZE)),
lr-all-nodes (identity (offset (add-addr (LR-F-ADDR,
LR-NODE-SIZE))),
tag ('addr,
cons (identity (LR-HEAP-NAME),
identity (LR-NODE-SIZE)
* *heap-size*))),
lr-init-data-seg (*heap-size*),
tag ('addr,
cons (identity (LR-HEAP-NAME),
identity (LR-NODE-SIZE) * *heap-size*)))

THEOREM: lr-proper-free-listp-lr-init-data-seg
(*heap-size* $\not<$ 2) → lr-proper-free-listp (lr-init-data-seg (*heap-size*))

THEOREM: definedp-lr-heap-name-lr-init-data-seg
definedp (identity (LR-HEAP-NAME), lr-init-data-seg (*heap-size*))

THEOREM: lr-proper-p-areasp-lr-heap-name-lr-init-data-seg
lr-proper-p-areasp (lr-init-data-seg (*heap-size*))

THEOREM: lr-proper-p-areasp-car-lr-init-data-seg-table
lr-proper-p-areasp (*data-seg*)
→ lr-proper-p-areasp (car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))

THEOREM: lr-proper-p-areasp-car-lr-data-seg-table-list
lr-proper-p-areasp (*data-seg*)
→ lr-proper-p-areasp (car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))

THEOREM: definedp-table-definedp-car-lr-init-data-seg-table
definedp (*name*, car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))
= definedp (*name*, *data-seg*)

THEOREM: all-p-objects-lookup-cons-table
(all-p-objects-lookup (*list*, *table*, *p*) ∧ p-objectp (*y*, *p*))
→ all-p-objects-lookup (*list*, cons (cons (*x*, *y*), *table*), *p*)

THEOREM: p-objectp-opener-alt-lr-proper-free-listp
(lr-proper-free-listp (p-data-segment (*p*))
 ∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*))
 ∧ (*data-seg* = p-data-segment (*p*)))
→ p-objectp (fetch (identity (LR-FP-ADDR), *data-seg*), *p*)

THEOREM: p-objectp-lookup-deposit-a-list
p-objectp (*object*,
 p-state (*pc*,
 ctrl-stk,
 temp-stk,
 prog-seg,
 data-seg,
 max-ctrl-stk-size,
 max-temp-stk-size,
 word-size,
 psw))
→ p-objectp (*object*,
 p-state (*pc*,
 ctrl-stk,
 temp-stk,
 prog-seg,
 deposit-a-list (*stuff*, *addr*, *data-seg*),
 max-ctrl-stk-size,
 max-temp-stk-size,
 word-size,
 psw))

THEOREM: all-p-objects-lookup-deposit-a-list
all-p-objects-lookup (*list*,
 table,
 p-state (*pc*,
 ctrl-stk,
 temp-stk,
 prog-seg,
 data-seg,
 max-ctrl-stk-size,
 max-temp-stk-size,
 word-size,
 psw))
→ all-p-objects-lookup (*list*,
 table,
 p-state (*pc*,

ctrl-stk,
temp-stk,
prog-seg,
deposit-a-list (*stuff*, *addr*, *data-seg*),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: p-objectp-lookup-deposit
p-objectp (*object*,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))
→ p-objectp (*object*,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
deposit (*anything*, *addr*, *data-seg*),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: all-p-objects-lookup-deposit
all-p-objects-lookup (*list*,
table,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))
→ all-p-objects-lookup (*list*,

table,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
deposit (*anything*, *addr*, *data-seg*),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: definedp-name-p-objectp-tag-0-lr-proper-p-areasp
lr-proper-p-areasp (*data-seg*)
→ (p-objectp (list (`'addr`, cons (*name*, `0`)),
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))
= definedp (*name*, *data-seg*))

THEOREM: all-p-objects-lookup-lr-compile-quote
(all-p-objects-lookup (*list*,
table,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))
∧ lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ all-p-objects-lookup (*list*,
cdr (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)),
p-state (*pc*,

ctrl-stk,
temp-stk,
prog-seg,
car (lr-compile-quote (*flag*,
object,
data-seg,
table)),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: all-p-objects-lookup-lr-data-seg-table-body
(all-p-objects-lookup (*list*,
table,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))
∧ lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ all-p-objects-lookup (*list*,
cdr (lr-data-seg-table-body (*flag*,
body,
data-seg,
table)),
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
car (lr-data-seg-table-body (*flag*,
body,
data-seg,
table)),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,

psw))

THEOREM: all-p-objects-lookup-lr-data-seg-table-list
(all-p-objects-lookup (*list*,
table,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))
∧ lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ all-p-objects-lookup (*list*,
cdr (lr-data-seg-table-list (*progs*, *data-seg*, *table*)),
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
car (lr-data-seg-table-list (*progs*,
data-seg,
table)),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: p-objectp-lookup-lr-init-data-seg-table
(p-objectp (*object*,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))
∧ lr-proper-free-listp (*data-seg*)

$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*)))
$\rightarrow$ p-objectp (*object*,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
car (lr-init-data-seg-table (*params*, *data-seg*, *table*)),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: assoc-definedp-table-lr-compile-quote
definedp (*object1*, *table*)
$\rightarrow$ (assoc (*object1*, cdr (lr-compile-quote (*flag*, *object2*, *data-seg*, *table*)))
$=$ assoc (*object1*, *table*))

THEOREM: assoc-definedp-table-lr-init-data-seg-table
definedp (*object*, *table*)
$\rightarrow$ (assoc (*object*, cdr (lr-init-data-seg-table (*params*, *data-seg*, *table*)))
$=$ assoc (*object*, *table*))

THEOREM: definedp-table-lr-compile-quote-self
(*flag* $\neq$ `'list`)
$\rightarrow$ definedp (*object*, cdr (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))

THEOREM: lr-s-similar-const-table-lr-good-pointerp-opener
(lr-s-similar-const-table (*table*, *data-seg*) $\wedge$ definedp (*object*, *table*))
$\rightarrow$ ((type (cdr (assoc (*object*, *table*))) = `'addr`)
$\wedge$ (cddr (cdr (assoc (*object*, *table*))) = **nil**)
$\wedge$ listp (cdr (assoc (*object*, *table*)))
$\wedge$ adpp (untag (cdr (assoc (*object*, *table*))), *data-seg*)
$\wedge$ lr-boundary-nodep (cdr (assoc (*object*, *table*)))
$\wedge$ (area-name (cdr (assoc (*object*, *table*)))
$=$ identity (LR-HEAP-NAME))
$\wedge$ (type (fetch (add-addr (cdr (assoc (*object*, *table*)),
identity (LR-REF-COUNT-OFFSET)),
data-seg))
$=$ `'nat`))

THEOREM: lr-s-similar-const-table-deposit-lr-fp-addr
(adpp (untag (LR-FP-ADDR), *data-seg*)
$\wedge$ lr-s-similar-const-table (*table*, *data-seg*))

→ lr-s-similar-const-table (*table*,
deposit (*anything*,
identity (LR-FP-ADDR),
data-seg))

THEOREM: adpp-fetch-lr-fp-addr-car-lr-compile-quote
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ adpp (untag (fetch (identity (LR-FP-ADDR),
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))),
data-seg)

DEFINITION:
lr-good-pointerp-tablep (*table*, *data-seg*)
= **if** listp (*table*)
then lr-good-pointerp (cdar (*table*), *data-seg*)
∧ lr-good-pointerp-tablep (cdr (*table*), *data-seg*)
else t endif

THEOREM: lr-good-pointerp-tablep-definedp-table
(lr-good-pointerp-tablep (*table*, *data-seg*) ∧ definedp (*object*, *table*))
→ lr-good-pointerp (cdr (assoc (*object*, *table*)), *data-seg*)

THEOREM: lr-proper-free-listp-opener-2-area-name-alt
(lr-proper-free-listp (*data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (car (untag (fetch (identity (LR-FP-ADDR), *data-seg*))) = LR-HEAP-NAME)

THEOREM: lr-good-pointerp-tablep-deposit-free-ptr
lr-good-pointerp-tablep (*table*,
deposit (*anything*, identity (LR-FP-ADDR), *data-seg*))
= lr-good-pointerp-tablep (*table*, *data-seg*)

THEOREM: add1-lr-boundary-nodep
(lr-boundary-nodep (*addr1*) ∧ lr-boundary-nodep (*addr2*))
→ ((offset (*addr1*) = (1 + offset (*addr2*))) = **f**)

THEOREM: lr-boundary-offsetp-plus
lr-boundary-offsetp (*n*)
→ (lr-boundary-offsetp (*m* + *n*) = lr-boundary-offsetp (*m*))

THEOREM: add1-add1-lr-boundary-nodep
(lr-boundary-nodep (*addr1*) ∧ lr-boundary-nodep (*addr2*))
→ ((offset (*addr1*) = (1 + (1 + offset (*addr2*)))) = **f**)

THEOREM: lr-good-pointerp-tablep-deposit-a-list
(lr-good-pointerp-tablep (*table*, *data-seg*)
 ∧ (offset (*addr*) ∈ **N**)
 ∧ lr-boundary-nodep (*addr*)
 ∧ (type (*ref-count*) = 'nat)
 ∧ (untag (*ref-count*) ∈ **N**)
 ∧ (area-name (*addr*) = LR-HEAP-NAME)
 ∧ (type (*tag*) = 'nat))
→ lr-good-pointerp-tablep (*table*,
deposit-a-list (list (*tag*, *ref-count*, *x*, *y*),
addr,
data-seg))

THEOREM: lr-good-pointerp-table-cons
((type (*addr*) = 'addr)
 ∧ (cddr (*addr*) = **nil**)
 ∧ listp (*addr*)
 ∧ adpp (untag (*addr*), *data-seg*)
 ∧ lr-boundary-nodep (*addr*)
 ∧ (area-name (*addr*) = LR-HEAP-NAME))
→ (lr-good-pointerp-tablep (cons (cons (*object*, *addr*), *table*), *data-seg*)
 = ((type (fetch (add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)),
data-seg))
 = 'nat)
 ∧ lr-good-pointerp-tablep (*table*, *data-seg*)))

THEOREM: lr-proper-free-listp-length-sub1-not-lessp
(lr-proper-free-listp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ definedp (LR-HEAP-NAME, *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ ((length (cdr (assoc (identity (LR-HEAP-NAME), *data-seg*))) − 1)
 ≮ offset (fetch (identity (LR-FP-ADDR),
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))))

THEOREM: lr-minimum-heapp-lr-compile-quote
(lr-proper-free-listp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ definedp (LR-HEAP-NAME, *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (lr-minimum-heapp (car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))
 = lr-minimum-heapp (*data-seg*))

DEFINITION:
s-heap-reqs (*flag*, *object*, *data-seg*, *table*)

```
=  if flag = 'list
   then if listp(object)
        then let pair be lr-compile-quote(t,
                                          car(object),
                                          data-seg,
                                          table)
             in
             s-heap-reqs(t, car(object), data-seg, table)
             +  s-heap-reqs('list,
                            cdr(object),
                            car(pair),
                            cdr(pair)) endlet
        else 0 endif
   elseif definedp(object, table) then 0
   elseif listp(object)
   then 1 + s-heap-reqs('list,
                        list(car(object), cdr(object)),
                        data-seg,
                        table)
   elseif object ∈ N then 1
   elseif truep(object) then 1
   else 0 endif
```

EVENT: Disable s-heap-reqs.

DEFINITION:

```
s-heap-reqs-body(flag, expr, data-seg, table)
=  if flag = 'list
   then if listp(expr)
        then let dst1 be lr-data-seg-table-body(t,
                                                car(expr),
                                                data-seg,
                                                table)
             in
             s-heap-reqs-body(t, car(expr), data-seg, table)
             +  s-heap-reqs-body('list,
                                 cdr(expr),
                                 car(dst1),
                                 cdr(dst1)) endlet
        else 0 endif
   elseif listp(expr)
   then if (car(expr) = S-TEMP-FETCH)
          ∨  (car(expr) = S-TEMP-EVAL)
          ∨  (car(expr) = S-TEMP-TEST)
```

```
        then s-heap-reqs-body (t, cadr (expr), data-seg, table)
        elseif car (expr) = 'quote
        then s-heap-reqs (t, cadr (expr), data-seg, table)
        else s-heap-reqs-body ('list, cdr (expr), data-seg, table) endif
    else 0 endif
```

DEFINITION:

```
s-heap-reqs-list (progs, data-seg, table)
=   if listp (progs)
    then s-heap-reqs-body (t, s-body (car (progs)), data-seg, table)
         +   s-heap-reqs-list (cdr (progs),
                               car (lr-data-seg-table-body (t,
                                                            s-body (car (progs)),
                                                            data-seg,
                                                            table)),
                               cdr (lr-data-seg-table-body (t,
                                                            s-body (car (progs)),
                                                            data-seg,
                                                            table)))
    else 0 endif
```

DEFINITION:

```
s-init-heap-reqs (params, data-seg, table)
=   if listp (params)
    then s-heap-reqs (t, cdar (params), data-seg, table)
         +   s-init-heap-reqs (cdr (params),
                               car (lr-compile-quote (t,
                                                      cdar (params),
                                                      data-seg,
                                                      table)),
                               cdr (lr-compile-quote (t,
                                                      cdar (params),
                                                      data-seg,
                                                      table)))
    else 0 endif
```

DEFINITION:

```
s-total-heap-reqs (progs, params, heap-size)
=   let init-ds-table1 be lr-compile-quote ('list,
                                            list (t, 0),
                                            lr-init-data-seg (heap-size),
                                            list (cons (f, LR-F-ADDR)))
    in
    let init-ds-table2 be lr-init-data-seg-table (params,
                                                  car (init-ds-table1),
```

cdr (*init-ds-table1*))
in
2
\+ s-heap-reqs ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
list (cons (**f**, LR-F-ADDR)))
\+ s-init-heap-reqs (*params*,
car (*init-ds-table1*),
cdr (*init-ds-table1*))
\+ s-heap-reqs-list (*progs*,
car (*init-ds-table2*),
cdr (*init-ds-table2*)) **endlet endlet**

DEFINITION:
s-ws-reqs (*flag*, *object*, *data-seg*, *table*)
= **if** *flag* = 'list
then if listp (*object*)
then let *pair* **be** lr-compile-quote (**t**,
car (*object*),
data-seg,
table)
in
max (s-ws-reqs (**t**, car (*object*), *data-seg*, *table*),
s-ws-reqs ('list,
cdr (*object*),
car (*pair*),
cdr (*pair*))) **endlet**
else 0 **endif**
elseif definedp (*object*, *table*) **then** 0
elseif listp (*object*)
then max (log (2, LR-CONS-TAG),
s-ws-reqs ('list,
list (car (*object*), cdr (*object*)),
data-seg,
table))
elseif *object* ∈ **N** **then** max (log (2, LR-ADD1-TAG), log (2, *object*))
elseif truep (*object*) **then** log (2, LR-TRUE-TAG)
elseif falsep (*object*) **then** log (2, LR-FALSE-TAG)
else 0 **endif**

EVENT: Disable s-ws-reqs.

DEFINITION:

```
s-ws-reqs-body (flag, expr, data-seg, table)
=  if flag = 'list
   then if listp (expr)
        then let dst1 be lr-data-seg-table-body (t,
                                                 car (expr),
                                                 data-seg,
                                                 table)
             in
             max (s-ws-reqs-body (t, car (expr), data-seg, table),
                  s-ws-reqs-body ('list,
                                  cdr (expr),
                                  car (dst1),
                                  cdr (dst1))) endlet
        else 0 endif
   elseif listp (expr)
   then if (car (expr) = S-TEMP-FETCH)
           ∨  (car (expr) = S-TEMP-EVAL)
           ∨  (car (expr) = S-TEMP-TEST)
        then s-ws-reqs-body (t, cadr (expr), data-seg, table)
        elseif car (expr) = 'quote
        then s-ws-reqs (t, cadr (expr), data-seg, table)
        else s-ws-reqs-body ('list, cdr (expr), data-seg, table) endif
   else 0 endif
```

DEFINITION:

```
s-ws-reqs-list (progs, data-seg, table)
=  if listp (progs)
   then max (s-ws-reqs-body (t, s-body (car (progs)), data-seg, table),
             s-ws-reqs-list (cdr (progs),
                             car (lr-data-seg-table-body (t,
                                                          s-body (car (progs)),
                                                          data-seg,
                                                          table)),
                             cdr (lr-data-seg-table-body (t,
                                                          s-body (car (progs)),
                                                          data-seg,
                                                          table))))
   else 0 endif
```

DEFINITION:

```
s-init-ws-reqs (params, data-seg, table)
=  if listp (params)
   then max (s-ws-reqs (t, cdar (params), data-seg, table),
             s-init-ws-reqs (cdr (params),
```

```
                                car (lr-compile-quote (t,
                                                       cdar (params),
                                                       data-seg,
                                                       table)),
                                cdr (lr-compile-quote (t,
                                                       cdar (params),
                                                       data-seg,
                                                       table))))
    else 0 endif
```

DEFINITION:

```
s-total-ws-reqs (progs, params, heap-size)
= let init-ds-table1 be lr-compile-quote ('list,
                                          list (t, 0),
                                          lr-init-data-seg (heap-size),
                                          list (cons (f, LR-F-ADDR)))
  in
  let init-ds-table2 be lr-init-data-seg-table (params,
                                                car (init-ds-table1),
                                                cdr (init-ds-table1))
  in
  max (s-ws-reqs ('list,
                  list (f, t, 0),
                  lr-init-data-seg (heap-size),
                  nil),
       max (s-init-ws-reqs (params,
                            car (init-ds-table1),
                            cdr (init-ds-table1)),
            max (s-ws-reqs-list (progs,
                                 car (init-ds-table2),
                                 cdr (init-ds-table2)),
                 S-MAX-SUBR-REQS))) endlet endlet
```

DEFINITION:

```
s-restricted-objectp (flag, object)
= if flag = 'list
  then if listp (object)
       then s-restricted-objectp (t, car (object))
            ∧  s-restricted-objectp ('list, cdr (object))
       else t endif
  elseif object = t then t
  elseif object = f then t
  elseif listp (object)
  then s-restricted-objectp ('list, list (car (object), cdr (object)))
```

elseif *object* ∈ **N** **then t**
else f endif

DEFINITION:
s-data-seg-body-restrictedp (*flag*, *expr*)
= **if** *flag* = `'list`
then if listp (*expr*)
then s-data-seg-body-restrictedp (**t**, car (*expr*))
∧ s-data-seg-body-restrictedp (`'list`, cdr (*expr*))
else t endif
elseif listp (*expr*)
then if (car (*expr*) = S-TEMP-FETCH)
∨ (car (*expr*) = S-TEMP-EVAL)
∨ (car (*expr*) = S-TEMP-TEST)
then s-data-seg-body-restrictedp (**t**, cadr (*expr*))
elseif car (*expr*) = `'quote`
then s-restricted-objectp (**t**, cadr (*expr*))
else s-data-seg-body-restrictedp (`'list`, cdr (*expr*)) **endif**
else t endif

DEFINITION:
s-data-seg-list-restrictedp (*progs*)
= **if** listp (*progs*)
then s-data-seg-body-restrictedp (**t**, s-body (car (*progs*)))
∧ s-data-seg-list-restrictedp (cdr (*progs*))
else t endif

DEFINITION:
s-init-data-seg-restrictedp (*params*)
= **if** listp (*params*)
then s-restricted-objectp (**t**, cdar (*params*))
∧ s-init-data-seg-restrictedp (cdr (*params*))
else t endif

DEFINITION:
s-restrictedp (*progs*, *params*)
= (s-init-data-seg-restrictedp (*params*)
∧ s-data-seg-list-restrictedp (*progs*))

THEOREM: lr-minimum-heapp-not-equal-length-1
lr-minimum-heapp (*data-seg*)
→ (length (cdr (assoc (identity (LR-HEAP-NAME), *data-seg*))) ≠ `1`)

THEOREM: lr-count-free-nodes-at-most
length (*node-list*) ≮ lr-count-free-nodes (*addr*, *node-list*, *data-seg*)

THEOREM: lr-proper-free-listp-lr-count-free-nodes-max-addr
(lr-proper-free-listp (*data-seg*)
 ∧ definedp (LR-HEAP-NAME, *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*))
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ (length (cdr (assoc (LR-HEAP-NAME, *data-seg*)))
  = (1 + offset (fetch (LR-FP-ADDR, *data-seg*))))
 ∧ (*max-addr* = lr-max-node (*data-seg*)))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR), *data-seg*),
  lr-free-list-nodes (*max-addr*, *data-seg*),
  data-seg)
  = 0)

EVENT: Disable lr-count-free-nodes-at-most.

THEOREM: lr-count-free-nodes-deposit-a-list-lr-nodep
((type (*addr1*) = 'addr)
 ∧ (cddr (*addr1*) = **nil**)
 ∧ listp (*addr1*)
 ∧ adpp (untag (*addr1*), *data-seg*)
 ∧ lr-boundary-nodep (*addr1*)
 ∧ (area-name (*addr1*) = LR-HEAP-NAME)
 ∧ (type (*ref-count*) = 'nat)
 ∧ (untag (*ref-count*) ∈ **N**)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*))
 ∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
 ∧ lr-node-listp (*node-list*, *data-seg*)
 ∧ (*addr1* ∉ *node-list*))
→ (lr-count-free-nodes (*addr2*,
  node-list,
  deposit-a-list (list (*x*, *ref-count*, *y*, *z*),
  addr1,
  data-seg))
  = lr-count-free-nodes (*addr2*, delete (*addr1*, *node-list*), *data-seg*))

THEOREM: lr-proper-free-listp-member-free-addr-lr-free-list-nodes
(adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*))
 ∧ (*max-addr* = lr-max-node (*data-seg*))
 ∧ (offset (fetch (LR-FP-ADDR, *data-seg*))
  < (length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) − 1))
 ∧ lr-proper-free-listp (*data-seg*))
→ (fetch (identity (LR-FP-ADDR), *data-seg*)
  ∈ lr-free-list-nodes (*max-addr*, *data-seg*))

THEOREM: lr-count-free-nodes-lr-compile-quote-s-heap-reqs-help1
(adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*))
 ∧ (offset (fetch (LR-FP-ADDR, *data-seg*))
  < (length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) − 1))
 ∧ lr-proper-free-listp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ (*max-addr* = lr-max-node (*data-seg*)))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR), *data-seg*),
 lr-free-list-nodes (*max-addr*, *data-seg*),
 data-seg)
  = (1 + lr-count-free-nodes (fetch (add-addr (fetch (identity (LR-FP-ADDR),
 data-seg),
 identity (LR-REF-COUNT-OFFSET)),
 data-seg),
 delete (fetch (identity (LR-FP-ADDR),
 data-seg),
 lr-free-list-nodes (*max-addr*,
 data-seg)),
 data-seg)))

THEOREM: lr-proper-free-listp-lr-count-free-nodes-max-addr-alt
(lr-proper-free-listp (*data-seg*)
 ∧ definedp (LR-HEAP-NAME, *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*))
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ (offset (fetch (LR-FP-ADDR, *data-seg*))
  ≮ (length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) − 1))
 ∧ (*max-addr* = lr-max-node (*data-seg*)))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR), *data-seg*),
 lr-free-list-nodes (*max-addr*, *data-seg*),
 data-seg)
  = `0`)

THEOREM: s-heap-reqs-object-t
((*flag* ≠ `'list`) ∧ (¬ definedp (`t`, *table*)))
→ (s-heap-reqs (*flag*, `t`, *data-seg*, *table*) = `1`)

THEOREM: lessp-lr-boundary-offsetp-nodep-plus-node-size-fact-1
(lr-boundary-offsetp (*offset*) ∧ lr-boundary-nodep (*addr*))
→ ((*offset* < (identity (LR-NODE-SIZE) + offset (*addr*)))
  = (offset (*addr*) ≮ *offset*))

THEOREM: lr-count-free-nodes-lr-compile-quote-s-heap-reqs
(lr-proper-free-listp (*data-seg*)

$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*)))
$\rightarrow$ ((lr-count-free-nodes (fetch (LR-FP-ADDR,
car (lr-compile-quote (*flag*,
object,
data-seg,
table))),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (lr-compile-quote (*flag*,
object,
data-seg,
table))),
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))
$+$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*))
$=$ lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg))

THEOREM: lr-compile-quote-lr-good-pointerp-tablep-help-1
let *ccar* **be** lr-compile-quote (**t**, *object*, *data-seg*, *table*)
in
(lr-proper-free-listp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR), *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ (s-heap-reqs (**t**, *object*, *data-seg*, *table*) $+$ *x*)))
$\rightarrow$ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR), car (*ccar*)),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (*ccar*)),
car (*ccar*))
$\not<$ *x*) **endlet**

THEOREM: s-heap-reqs-flag-list-nil-opener
s-heap-reqs ('list, **nil**, *data-seg*, *table*) = 0

THEOREM: lr-compile-quote-flag-list-nil-opener
lr-compile-quote ('list, **nil**, *data-seg*, *table*) = cons (*data-seg*, *table*)

THEOREM: s-heap-reqs-flag-list-cons-opener
s-heap-reqs ('list, cons (*x*, *y*), *data-seg*, *table*)
= (s-heap-reqs (**t**, *x*, *data-seg*, *table*)
+ s-heap-reqs ('list,
y,
car (lr-compile-quote (**t**, *x*, *data-seg*, *table*)),
cdr (lr-compile-quote (**t**, *x*, *data-seg*, *table*))))

THEOREM: lr-compile-quote-flag-list-cons-opener
lr-compile-quote ('list, cons (*x*, *y*), *data-seg*, *table*)
= lr-compile-quote ('list,
y,
car (lr-compile-quote (**t**, *x*, *data-seg*, *table*)),
cdr (lr-compile-quote (**t**, *x*, *data-seg*, *table*)))

THEOREM: lr-compile-quote-lr-good-pointerp-tablep-help-2
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*))
∧ lr-minimum-heapp (*data-seg*)
∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
≠ 0)
∧ (lr-count-free-nodes (fetch (add-addr (fetch (LR-FP-ADDR, *data-seg*),
LR-REF-COUNT-OFFSET),
data-seg),
delete (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg)),
data-seg)
≮ s-heap-reqs ('list,
list (car (*object*), cdr (*object*)),
data-seg,
table))
∧ listp (*object*))
→ ((length (cdr (assoc (identity (LR-HEAP-NAME), *data-seg*))) − 1)

$\not<$ (offset (fetch (identity (LR-FP-ADDR),
car (lr-compile-quote ('list,
list (car (*object*),
cdr (*object*)),
data-seg,
table))))
$+$ identity (LR-NODE-SIZE)))

EVENT: Disable s-heap-reqs-flag-list-cons-opener.

EVENT: Disable lr-compile-quote-flag-list-cons-opener.

THEOREM: lr-compile-quote-lr-good-pointerp-tablep-help-3
(lr-proper-free-listp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ 1))
$\rightarrow$ ((length (cdr (assoc (identity (LR-HEAP-NAME), *data-seg*))) $-$ 1)
$\not<$ (offset (fetch (identity (LR-FP-ADDR), *data-seg*))
$+$ identity (LR-NODE-SIZE)))

THEOREM: lr-compile-quote-lr-good-pointerp-tablep
(lr-proper-free-listp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-restricted-objectp (*flag*, *object*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*))
$\wedge$ definedp (**f**, *table*))
$\rightarrow$ lr-good-pointerp-tablep (cdr (lr-compile-quote (*flag*,
object,

data-seg,
table)),
car (lr-compile-quote (*flag*,
object,
data-seg,
table)))

THEOREM: lr-nodep-car-lr-compile-quote
lr-nodep (*addr*, *data-seg*)
→ lr-nodep (*addr*, car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))

THEOREM: lr-proper-free-listp-opener-2-lr-nodep
(lr-proper-free-listp (*data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*))
∧ (*addr* = fetch (identity (LR-FP-ADDR), *data-seg*)))
→ lr-nodep (*addr*, *data-seg*)

THEOREM: lr-nodep-deposit-a-list
lr-nodep (*addr1*, *data-seg*)
→ lr-nodep (*addr1*, deposit-a-list (*list*, *addr2*, *data-seg*))

DEFINITION:
induct-hint-16 (*object*, *list*, *data-seg*, *table*)
= **if** *list* ≃ **nil** **then t**
elseif *object* = car (*list*) **then t**
else induct-hint-16 (*object*,
cdr (*list*),
car (lr-compile-quote (**t**,
car (*list*),
data-seg,
table)),
cdr (lr-compile-quote (**t**,
car (*list*),
data-seg,
table))) **endif**

THEOREM: definedp-object-cdr-lr-compile-quote-list
(*object* ∈ *list*)
→ definedp (*object*, cdr (lr-compile-quote (`'list`, *list*, *data-seg*, *table*)))

THEOREM: lr-good-pointerp-cdr-assoc-car-lr-compile-quote-list
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)

$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-restricted-objectp (`'list`, *object-list*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\neq$ `0`)
$\wedge$ ((lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg) $-$ 1)
$\not<$ s-heap-reqs (`'list`, *object-list*, *data-seg*, *table*))
$\wedge$ definedp (**f**, *table*)
$\wedge$ (*object* $\in$ *object-list*))
$\rightarrow$ lr-good-pointerp (cdr (assoc (*object*,
cdr (lr-compile-quote (`'list`,
object-list,
data-seg,
table)))),
car (lr-compile-quote (`'list`,
object-list,
data-seg,
table)))

THEOREM: lr-good-pointerp-deposit-non-ref-not-good-pointerp
(lr-boundary-nodep (*addr*)
$\wedge$ adpp (untag (*addr*), *data-seg*)
$\wedge$ (area-name (*addr*) = `'heap`)
$\wedge$ (type (*addr*) = `'addr`)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ ($n <$ LR-NODE-SIZE)
$\wedge$ ($n \neq$ LR-REF-COUNT-OFFSET)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-good-pointerp (*good-pointer*, *data-seg*)
$\wedge$ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) $\neq$ `'nat`))
$\rightarrow$ lr-good-pointerp (*good-pointer*, deposit (x, add-addr (*addr*, n), *data-seg*))

THEOREM: lr-good-pointerp-deposit-ref-count-not-good-pointerp
(lr-boundary-nodep (*addr*)
$\wedge$ adpp (untag (*addr*), *data-seg*)
$\wedge$ (area-name (*addr*) = `'heap`)

$\wedge$ (type (*addr*) = 'addr)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ (type (*x*) = 'nat)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-good-pointerp (*good-pointer*, *data-seg*)
$\wedge$ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) $\neq$ 'nat))
$\rightarrow$ lr-good-pointerp (*good-pointer*,
deposit (*x*,
add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)),
data-seg))

THEOREM: lr-good-pointerp-deposit-non-add-addr-not-good-pointerp
(lr-boundary-nodep (*addr*)
$\wedge$ adpp (untag (*addr*), *data-seg*)
$\wedge$ (area-name (*addr*) = 'heap)
$\wedge$ (type (*addr*) = 'addr)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-good-pointerp (*good-pointer*, *data-seg*)
$\wedge$ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) $\neq$ 'nat))
$\rightarrow$ lr-good-pointerp (*good-pointer*, deposit (*x*, *addr*, *data-seg*))

THEOREM: lr-check-numberp-addrp-deposit-a-list-cons-same-addr
((type (*addr*) = 'addr)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ adpp (untag (*addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ (area-name (*addr*) = 'heap)
$\wedge$ (offset (*addr*) $\not<$ (LR-NODE-SIZE + offset (LR-F-ADDR)))
$\wedge$ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*))
= 'addr)
$\wedge$ lr-good-pointerp (*good-pointer*, *data-seg*)
$\wedge$ definedp ('heap, *data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ (type (*ref-count*) = 'nat)
$\wedge$ (untag (*ref-count*) $\in$ **N**)
$\wedge$ (type (*tagged-number*) = 'nat)
$\wedge$ (untag (*tagged-number*) $\in$ **N**))
$\rightarrow$ lr-check-numberp-addrp (*addr*,
deposit-a-list (list (identity (tag ('nat,
LR-ADD1-TAG)),

ref-count,
tagged-number,
good-pointer),
addr,
data-seg))

THEOREM: lr-proper-heapp-nodep-deposit-a-list-numberp
(lr-nodep (*max-addr*, *data-seg*)
∧ lr-nodep (*addr*, *data-seg*)
∧ lr-proper-heapp-nodep (*max-addr*, *data-seg*)
∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) ≠ 'nat)
∧ lr-good-pointerp (*good-pointer*, *data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ (type (*ref-count*) = 'nat)
∧ (untag (*ref-count*) ∈ **N**)
∧ (type (*tagged-number*) = 'nat)
∧ (untag (*tagged-number*) ∈ **N**))
→ lr-proper-heapp-nodep (*max-addr*,
deposit-a-list (list (identity (tag ('nat,
LR-ADD1-TAG)),
ref-count,
tagged-number,
good-pointer),
addr,
data-seg))

THEOREM: lr-proper-heapp2-deposit-a-list-numberp
(lr-nodep (*max-addr*, *data-seg*)
∧ lr-nodep (*addr*, *data-seg*)
∧ lr-proper-heapp2 (*max-addr*, *data-seg*)
∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) ≠ 'nat)
∧ lr-good-pointerp (*good-pointer*, *data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ (type (*tag*) = 'nat)
∧ (untag (*tag*) ∈ **N**)
∧ (type (*tagged-number*) = 'nat)
∧ (untag (*tagged-number*) ∈ **N**))
→ lr-proper-heapp2 (*max-addr*,
deposit-a-list (list (identity (tag ('nat, LR-ADD1-TAG)),
tag,
tagged-number,
good-pointer),

addr,
data-seg))

THEOREM: lr-good-pointerp-lr-undef-addr
(lr-minimum-heapp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ lr-proper-heapp2 (lr-max-node (*data-seg*), *data-seg*)
∧ lr-nodep (lr-max-node (*data-seg*), *data-seg*))
→ lr-good-pointerp (identity (LR-UNDEF-ADDR), *data-seg*)

THEOREM: lr-proper-heapp-nodep-deposit-a-list-truep
(lr-nodep (*max-addr*, *data-seg*)
∧ lr-nodep (*addr*, *data-seg*)
∧ lr-proper-heapp-nodep (*max-addr*, *data-seg*)
∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) ≠ 'nat)
∧ lr-good-pointerp (*good-pointer1*, *data-seg*)
∧ lr-good-pointerp (*good-pointer2*, *data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ (type (*ref-count*) = 'nat)
∧ (untag (*ref-count*) ∈ **N**))
→ lr-proper-heapp-nodep (*max-addr*,
deposit-a-list (list (identity (tag ('nat,
LR-TRUE-TAG)),
ref-count,
good-pointer1,
good-pointer2),
addr,
data-seg))

THEOREM: lr-proper-heapp2-deposit-a-list-truep
(lr-nodep (*max-addr*, *data-seg*)
∧ lr-nodep (*addr*, *data-seg*)
∧ lr-proper-heapp2 (*max-addr*, *data-seg*)
∧ (type (fetch (add-addr (*addr*, LR-REF-COUNT-OFFSET), *data-seg*)) ≠ 'nat)
∧ lr-good-pointerp (*good-pointer1*, *data-seg*)
∧ lr-good-pointerp (*good-pointer2*, *data-seg*)
∧ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ (type (*tag*) = 'nat)
∧ (untag (*tag*) ∈ **N**))
→ lr-proper-heapp2 (*max-addr*,
deposit-a-list (list (identity (tag ('nat, LR-TRUE-TAG)),
tag,
good-pointer1,

good-pointer2),
addr,
data-seg))

THEOREM: lr-compile-quote-preserves-lr-proper-heapp2
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ lr-nodep (lr-max-node (*data-seg*), *data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*))
∧ lr-good-pointerp-tablep (*table*, *data-seg*)
∧ s-restricted-objectp (*flag*, *object*)
∧ lr-minimum-heapp (*data-seg*)
∧ lr-proper-heapp2 (lr-max-node (*data-seg*), *data-seg*)
∧ definedp (**f**, *table*)
∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*)))
→ lr-proper-heapp2 (lr-max-node (*data-seg*),
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))

THEOREM: plistp-pair-formals-with-addresses
plistp (pair-formals-with-addresses (*formals*, *table*))

THEOREM: strip-cars-pair-formals-with-addresses
strip-cars (pair-formals-with-addresses (*formals*, *table*))
= strip-cars (*formals*)

THEOREM: strip-cars-lr-make-initial-temps
strip-cars (lr-make-initial-temps (*temp-vars*)) = plist (*temp-vars*)

THEOREM: lr-s-similar-const-table-implies-lr-good-pointerp-tablep
lr-s-similar-const-table (*table*, *data-seg*)
→ lr-good-pointerp-tablep (*table*, *data-seg*)

THEOREM: lr-s-similar-const-table-deposit-cons
(lr-proper-heapp (*data-seg*)
∧ lr-s-similar-const-table (*table*, *data-seg*)
∧ lr-proper-p-areasp (*data-seg*))
→ lr-s-similar-const-table (*table*,
deposit-a-list (list (*x0*, *x1*, *x2*, *x3*),
fetch (identity (LR-FP-ADDR),
data-seg),
data-seg))

THEOREM: lr-valp-deposit-a-list-cons-cons
(listp (*object*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ lr-s-similar-const-table (*table*, *data-seg*)
 ∧ (type (*ref-count*) = 'nat)
 ∧ (untag (*ref-count*) ∈ **N**)
 ∧ definedp (car (*object*), *table*)
 ∧ definedp (cdr (*object*), *table*)
 ∧ definedp (LR-HEAP-NAME, *data-seg*)
 ∧ (*addr* = fetch (identity (LR-FP-ADDR), *data-seg*)))
→ lr-valp (*object*,
 addr,
 deposit-a-list (list (identity (tag ('nat, LR-CONS-TAG)),
 ref-count,
 cdr (assoc (car (*object*), *table*)),
 cdr (assoc (cdr (*object*), *table*))),
 addr,
 data-seg))

THEOREM: lr-valp-deposit-a-list-cons-numberp
((*object* ∈ **N**)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ (type (*ref-count*) = 'nat)
 ∧ (untag (*ref-count*) ∈ **N**)
 ∧ definedp (LR-HEAP-NAME, *data-seg*)
 ∧ (*addr* = fetch (identity (LR-FP-ADDR), *data-seg*)))
→ lr-valp (*object*,
 addr,
 deposit-a-list (list (identity (tag ('nat, LR-ADD1-TAG)),
 ref-count,
 tag ('nat, *object*),
 identity (LR-UNDEF-ADDR)),
 addr,
 data-seg))

THEOREM: lr-compile-quote-preserves-lr-valp
(lr-proper-p-areasp (*data-seg*)
 ∧ lr-good-pointerp-tablep (*table*, *data-seg*)
 ∧ s-restricted-objectp (*flag*, *object*)
 ∧ lr-minimum-heapp (*data-seg*)
 ∧ lr-nodep (lr-max-node (*data-seg*), *data-seg*)
 ∧ lr-proper-free-listp (*data-seg*)

$\wedge$ lr-proper-heapp2 (lr-max-node (*data-seg*), *data-seg*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*))
$\wedge$ lr-valp (*value*, *addr*, *data-seg*))
$\rightarrow$ lr-valp (*value*, *addr*, car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))

THEOREM: lr-compile-quote-preserves-lr-proper-heapp
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ s-restricted-objectp (*flag*, *object*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*)))
$\rightarrow$ lr-proper-heapp (car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))

THEOREM: lr-valp-deposit-a-list-cons-truep
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ (type (*ref-count*) = `'nat`)
$\wedge$ (untag (*ref-count*) $\in$ **N**)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ (*addr* = fetch (identity (LR-FP-ADDR), *data-seg*)))
$\rightarrow$ lr-valp (**t**,
addr,
deposit-a-list (list (identity (tag (`'nat`, LR-TRUE-TAG)),
ref-count,
identity (LR-UNDEF-ADDR),
identity (LR-UNDEF-ADDR)),
addr,
data-seg))

THEOREM: lr-s-similar-const-table-lr-compile-quote
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ s-restricted-objectp (*flag*, *object*)
$\wedge$ definedp (**f**, *table*)

$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*)))
$\rightarrow$ lr-s-similar-const-table (cdr (lr-compile-quote (*flag*,
object,
data-seg,
table)),
car (lr-compile-quote (*flag*,
object,
data-seg,
table)))

THEOREM: p-objectp-cdr-assoc-car-lr-compile-quote
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ s-restricted-objectp (*flag*, *object*)
$\wedge$ (*word-size* $\not<$ s-ws-reqs (*flag*, *object*, *data-seg*, *table*))
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*))
$\wedge$ (*flag* $\neq$ `'list`))
$\rightarrow$ p-objectp (cdr (assoc (*object*,
cdr (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))),
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: lr-count-free-nodes-s-init-heap-reqs
let *ccar* **be** lr-compile-quote (**t**, *object*, *data-seg*, *table*)
in
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)

$\wedge$ lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ (s-heap-reqs (**t**, *object*, *data-seg*, *table*)
$+$ s-init-heap-reqs (*params*, car (*ccar*), cdr (*ccar*)))))
$\rightarrow$ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR), car (*ccar*)),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (*ccar*)),
car (*ccar*))
$\not<$ s-init-heap-reqs (*params*, car (*ccar*), cdr (*ccar*))) **endlet**

THEOREM: all-p-objects-lookup-strip-cdrs-lr-init-data-seg-table
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ definedp (**t**, *table*)
$\wedge$ s-init-data-seg-restrictedp (*params*)
$\wedge$ (*word-size* $\not<$ s-init-ws-reqs (*params*, *data-seg*, *table*))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-init-heap-reqs (*params*, *data-seg*, *table*)))
$\rightarrow$ all-p-objects-lookup (strip-cdrs (*params*),
cdr (lr-init-data-seg-table (*params*, *data-seg*, *table*)),
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
car (lr-init-data-seg-table (*params*,
data-seg,
table)),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: lr-minimum-heapp-lr-init-data-seg
(*heap-size* $\not<$ 4) $\rightarrow$ lr-minimum-heapp (lr-init-data-seg (*heap-size*))

THEOREM: adpp-cons-heap-name-node-size-lr-init-data-seg
(*heap-size* $\not<$ 2)

$\rightarrow$ adpp (cons (identity (LR-HEAP-NAME), identity (LR-NODE-SIZE)
$*$ *heap-size*),
lr-init-data-seg (*heap-size*))

THEOREM: lr-check-f-addrp-lr-undef-addr-lr-init-data-seg
((offset (*addr*) = identity (offset (LR-UNDEF-ADDR)))
$\wedge$ (type (*addr*) = 'addr)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ (area-name (*addr*) = LR-HEAP-NAME)
$\wedge$ adpp (untag (*addr*), *data-seg*))
$\rightarrow$ lr-check-undef-addrp (*addr*, *data-seg*)

THEOREM: fetch-offset-lr-t-addr-ref-count-offset-compile-quote-t
((type (*addr*) = 'addr)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ (area-name (*addr*) = LR-HEAP-NAME)
$\wedge$ adpp (untag (*addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ ($\neg$ definedp (**t**, *table*))
$\wedge$ lr-proper-free-listp (*data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ (offset (fetch (LR-FP-ADDR, *data-seg*)) $<$ LR-MINIMUM-HEAP-SIZE)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ ((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) $-$ 1)
$\not<$ (offset (*addr*) + LR-NODE-SIZE)))
$\rightarrow$ (fetch (add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)),
car (lr-compile-quote (**t**, **t**, *data-seg*, *table*)))
= **if** offset (*addr*) = offset (fetch (LR-FP-ADDR, *data-seg*))
then identity (tag ('nat, 1))
else fetch (add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)),
data-seg) **endif**)

THEOREM: definedp-cdr-lr-compile-quote-t
definedp (*x*, cdr (lr-compile-quote (**t**, **t**, *data-seg*, *table*)))
= ((*x* = **t**) $\vee$ definedp (*x*, *table*))

THEOREM: fetch-lr-fp-addr-compile-quote-t
($\neg$ definedp (**t**, *table*))
$\wedge$ lr-proper-free-listp (*data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
$\wedge$ ((length (value (LR-HEAP-NAME, *data-seg*)) $-$ 1)
$\not<$ (offset (fetch (LR-FP-ADDR, *data-seg*)) + LR-NODE-SIZE))

$\wedge$ lr-proper-p-areasp (*data-seg*))
$\rightarrow$ (fetch (identity (LR-FP-ADDR), car (lr-compile-quote (**t**, **t**, *data-seg*, *table*)))
= fetch (add-addr (fetch (identity (LR-FP-ADDR), *data-seg*),
identity (LR-REF-COUNT-OFFSET)),
data-seg))

THEOREM: numberp-lessp-4-not-3-not-2-not-1-must-be-0
$((c \in \mathbf{N}) \wedge (c \neq \mathtt{3}) \wedge (c \neq \mathtt{2}) \wedge (c \neq \mathtt{1}) \wedge (c < \mathtt{4}))$
$\rightarrow$ $(c = \mathtt{0})$

EVENT: Disable fetch-offset-lr-t-addr-ref-count-offset-compile-quote-t.

THEOREM: lessp-minimum-heap-size-not-0-f-t-must-be-undef-alt-1-help
(lr-boundary-offsetp (*offset*)
$\wedge$ (*offset* $\in$ **N**)
$\wedge$ (*offset* $\neq$ offset (LR-UNDEF-ADDR))
$\wedge$ (*offset* $\neq$ offset (LR-F-ADDR))
$\wedge$ (*offset* $\neq$ offset (LR-T-ADDR))
$\wedge$ (*offset* $\neq$ offset (LR-0-ADDR)))
$\rightarrow$ (*offset* $\not<$ LR-MINIMUM-HEAP-SIZE)

THEOREM: lessp-minimum-heap-size-not-0-f-t-must-be-undef-alt-1
((offset (*addr*) < LR-MINIMUM-HEAP-SIZE)
$\wedge$ (type (*addr*) = `'addr`)
$\wedge$ (cddr (*addr*) = **nil**)
$\wedge$ listp (*addr*)
$\wedge$ (area-name (*addr*) = `'heap`)
$\wedge$ adpp (untag (*addr*), lr-init-data-seg (*heap-size*))
$\wedge$ lr-boundary-nodep (*addr*)
$\wedge$ (offset (*addr*) $\neq$ offset (LR-0-ADDR))
$\wedge$ (offset (*addr*) $\neq$ offset (LR-T-ADDR))
$\wedge$ (offset (*addr*) $\neq$ offset (LR-F-ADDR)))
$\rightarrow$ (fetch (*addr*, lr-init-data-seg (*heap-size*))
= identity (tag (`'nat`, LR-UNDEFINED-TAG)))

THEOREM: lessp-lr-boundary-offsetp-3
(lr-boundary-offsetp (*offset*) $\wedge$ (*offset* $\in$ **N**))
$\rightarrow$ ((*offset* < $\mathtt{3}$) = (*offset* = $\mathtt{0}$))

THEOREM: numberp-lessp-2-not-1-must-be-0
$((c \in \mathbf{N}) \wedge (c \neq \mathtt{1}) \wedge (c < \mathtt{2})) \rightarrow (c = \mathtt{0})$

THEOREM: not-lessp-difference-lr-boundary-offsetp-fact
((*offset* $\in$ **N**) $\wedge$ (*offset* < (LR-NODE-SIZE + offset (LR-F-ADDR))))
$\rightarrow$ (lr-boundary-offsetp (*offset*)

$$
\begin{array}{ll}
= & ((\mathit{offset} = \text{identity}\,(\text{offset}\,(\textsc{lr-f-addr}))) \\
 & \quad \vee \quad (\mathit{offset} = \text{identity}\,(\text{offset}\,(\textsc{lr-undef-addr})))))
\end{array}
$$

THEOREM: fetch-ref-count-lr-init-data-seg-free-list

$$
\begin{array}{ll}
(((\textsc{lr-node-size} * \mathit{heap\text{-}size}) \not< (\text{offset}\,(\mathit{addr}) + \textsc{lr-node-size})) & \\
\wedge & (\text{type}\,(\mathit{addr}) = \texttt{'addr}) \\
\wedge & (\text{cddr}\,(\mathit{addr}) = \textbf{nil}) \\
\wedge & \text{listp}\,(\mathit{addr}) \\
\wedge & (\text{area-name}\,(\mathit{addr}) = \textsc{lr-heap-name}) \\
\wedge & \text{listp}\,(\text{untag}\,(\mathit{addr})) \\
\wedge & \text{lr-boundary-nodep}\,(\mathit{addr}) \\
\wedge & (\mathit{heap\text{-}size} \not< \texttt{2})) \\
\rightarrow & (\text{fetch}\,(\text{add-addr}\,(\mathit{addr}, \text{identity}\,(\textsc{lr-ref-count-offset})), \\
 & \qquad \text{lr-init-data-seg}\,(\mathit{heap\text{-}size})) \\
 & = \quad \textbf{if}\ \text{offset}\,(\textsc{lr-f-addr}) < \text{offset}\,(\mathit{addr}) \\
 & \quad\ \ \textbf{then}\ \text{add-addr}\,(\mathit{addr}, \text{identity}\,(\textsc{lr-node-size})) \\
 & \quad\ \ \textbf{else}\ \text{identity}\,(\text{tag}\,(\texttt{'nat}, \texttt{1}))\ \textbf{endif})
\end{array}
$$

THEOREM: lessp-offset-lr-init-data-seg-adpp-untag-lessp-offset

$$
\begin{array}{ll}
(\text{adpp}\,(\text{untag}\,(\mathit{addr}), \text{lr-init-data-seg}\,(\mathit{heap\text{-}size})) & \\
\wedge & \text{lr-boundary-offsetp}\,(\mathit{offset}) \\
\wedge & \text{lr-boundary-offsetp}\,(\text{offset}\,(\mathit{addr})) \\
\wedge & (\mathit{offset} < \text{offset}\,(\mathit{addr})) \\
\wedge & (\text{area-name}\,(\mathit{addr}) = \textsc{lr-heap-name}) \\
\wedge & (\mathit{heap\text{-}size} \not< \texttt{2})) \\
\rightarrow & ((\mathit{offset} < (\text{identity}\,(\textsc{lr-node-size}) * \mathit{heap\text{-}size})) = \textbf{t})
\end{array}
$$

THEOREM: lr-count-free-nodes-lr-all-nodes

$$
\begin{array}{ll}
(\text{lr-boundary-nodep}\,(\mathit{addr}) & \\
\wedge & (\text{area-name}\,(\mathit{addr}) = \textsc{lr-heap-name}) \\
\wedge & \text{listp}\,(\text{untag}\,(\mathit{addr})) \\
\wedge & (\text{offset}\,(\mathit{addr}) \in \mathbf{N}) \\
\wedge & (\text{type}\,(\mathit{addr}) = \texttt{'addr}) \\
\wedge & (\text{cddr}\,(\mathit{addr}) = \textbf{nil}) \\
\wedge & \text{listp}\,(\mathit{addr}) \\
\wedge & \text{lr-boundary-nodep}\,(\mathit{max\text{-}addr}) \\
\wedge & (\text{area-name}\,(\mathit{max\text{-}addr}) = \textsc{lr-heap-name}) \\
\wedge & \text{adpp}\,(\text{untag}\,(\mathit{max\text{-}addr}), \text{lr-init-data-seg}\,(\mathit{heap\text{-}size})) \\
\wedge & (\text{type}\,(\mathit{max\text{-}addr}) = \texttt{'addr}) \\
\wedge & (\text{cddr}\,(\mathit{max\text{-}addr}) = \textbf{nil}) \\
\wedge & \text{listp}\,(\mathit{max\text{-}addr}) \\
\wedge & (\text{offset}\,(\textsc{lr-f-addr}) < \text{offset}\,(\mathit{addr})) \\
\wedge & (\mathit{heap\text{-}size} \not< \texttt{2})) \\
\rightarrow & (\text{lr-count-free-nodes}\,(\mathit{addr}, \\
 & \qquad\qquad \text{lr-all-nodes}\,(\text{offset}\,(\mathit{addr}),\ \mathit{max\text{-}addr}),
\end{array}
$$

lr-init-data-seg (*heap-size*))
= length (lr-all-nodes (offset (*addr*), *max-addr*)))

THEOREM: same-signature-car-lr-init-data-seg-table-help-1
let *comp-obj* **be** lr-compile-quote (**t**, *object*, *data-seg*, *table*)
in
(same-signature (car (*comp-obj*),
car (lr-init-data-seg-table (*params*,
car (*comp-obj*),
cdr (*comp-obj*))))
∧ lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ same-signature (*data-seg*,
car (lr-init-data-seg-table (*params*,
car (*comp-obj*),
cdr (*comp-obj*)))) **endlet**

THEOREM: same-signature-car-lr-init-data-seg-table
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ same-signature (*data-seg*,
car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))

THEOREM: lr-max-node-car-lr-init-data-seg-table
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (lr-max-node (car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))
= lr-max-node (*data-seg*))

THEOREM: s-heap-reqs-object-0
((*flag* ≠ 'list) ∧ (¬ definedp (0, *table*)))
→ (s-heap-reqs (*flag*, 0, *data-seg*, *table*) = 1)

THEOREM: lr-free-list-nodes-deposit-0
((type (*max-addr*) = 'addr)
∧ (cddr (*max-addr*) = **nil**)
∧ listp (*max-addr*)
∧ adpp (untag (*max-addr*), *data-seg*)
∧ lr-boundary-nodep (*max-addr*)

$\wedge$ (area-name (*max-addr*) = LR-HEAP-NAME)
$\wedge$ ($\neg$ definedp (0, *table*))
$\wedge$ lr-proper-free-listp (*data-seg*)
$\wedge$ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ ((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) $-$ 1)
$\not<$ (offset (fetch (LR-FP-ADDR, *data-seg*)) + LR-NODE-SIZE)))
$\rightarrow$ (lr-free-list-nodes (*max-addr*,
car (lr-compile-quote (**t**, 0, *data-seg*, *table*)))
= lr-free-list-nodes (*max-addr*,
deposit (identity (tag ('nat, 1)),
add-addr (fetch (identity (LR-FP-ADDR),
data-seg),
identity (LR-REF-COUNT-OFFSET)),
data-seg)))

THEOREM: lr-free-list-nodes-deposit-t
((type (*max-addr*) = 'addr)
$\wedge$ (cddr (*max-addr*) = **nil**)
$\wedge$ listp (*max-addr*)
$\wedge$ adpp (untag (*max-addr*), *data-seg*)
$\wedge$ lr-boundary-nodep (*max-addr*)
$\wedge$ (area-name (*max-addr*) = LR-HEAP-NAME)
$\wedge$ ($\neg$ definedp (**t**, *table*))
$\wedge$ lr-proper-free-listp (*data-seg*)
$\wedge$ adpp (untag (lr-max-node (*data-seg*)), *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ ((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) $-$ 1)
$\not<$ (offset (fetch (LR-FP-ADDR, *data-seg*)) + LR-NODE-SIZE)))
$\rightarrow$ (lr-free-list-nodes (*max-addr*,
car (lr-compile-quote (**t**, **t**, *data-seg*, *table*)))
= lr-free-list-nodes (*max-addr*,
deposit (identity (tag ('nat, 1)),
add-addr (fetch (identity (LR-FP-ADDR),
data-seg),
identity (LR-REF-COUNT-OFFSET)),
data-seg)))

THEOREM: fetch-lr-fp-addr-compile-quote-0
(($\neg$ definedp (0, *table*))
$\wedge$ ((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) $-$ 1)
$\not<$ (offset (fetch (LR-FP-ADDR, *data-seg*)) + LR-NODE-SIZE))

$\wedge$ definedp (area-name (LR-FP-ADDR), *data-seg*))
$\rightarrow$ (fetch (identity (LR-FP-ADDR),
car (lr-compile-quote (**t**, 0, *data-seg*, *table*)))
= fetch (add-addr (fetch (identity (LR-FP-ADDR), *data-seg*),
identity (LR-REF-COUNT-OFFSET)),
data-seg))

THEOREM: fetch-add-addr-ref-count-f-addr-lr-init-data-seg
(*heap-size* $\not<$ 2)
$\rightarrow$ (fetch (identity (add-addr (LR-F-ADDR, LR-REF-COUNT-OFFSET)),
lr-init-data-seg (*heap-size*))
= identity (tag ('nat, 1)))

THEOREM: lr-good-pointerp-tablep-f-lr-f-addr-lr-init-data-seg
(*heap-size* $\not<$ 4)
$\rightarrow$ lr-good-pointerp-tablep (list (cons (**f**, identity (LR-F-ADDR))),
lr-init-data-seg (*heap-size*))

THEOREM: lr-proper-heapp-nodep-lr-undef-addr-lr-init-data-seg
(*heap-size* $\not<$ 2)
$\rightarrow$ lr-proper-heapp-nodep (identity (LR-UNDEF-ADDR),
lr-init-data-seg (*heap-size*))

DEFINITION:
induct-hint-2 (*offset*)
= **if** *offset* $<$ offset (add-addr (LR-F-ADDR, LR-NODE-SIZE)) **then t**
else induct-hint-2 (*offset* $-$ LR-NODE-SIZE) **endif**

THEOREM: lr-boundary-offsetp-difference-lr-node-size
lr-boundary-offsetp (*offset*)
$\rightarrow$ lr-boundary-offsetp (*offset* $-$ identity (LR-NODE-SIZE))

THEOREM: fetch-lr-f-addr-lr-init-data-seg
(*heap-size* $\not<$ 2)
$\rightarrow$ (fetch (identity (LR-F-ADDR), lr-init-data-seg (*heap-size*))
= identity (tag ('nat, LR-FALSE-TAG)))

THEOREM: lr-proper-heapp-nodep-lr-init-data-seg-helper
((*offset* $\not<$ offset (LR-F-ADDR))
$\wedge$ (*heap-size* $\not<$ 2)
$\wedge$ (*offset* $\in$ **N**)
$\wedge$ lr-boundary-nodep (tag ('addr, cons (LR-HEAP-NAME, *offset*)))
$\wedge$ ((LR-NODE-SIZE $*$ *heap-size*) $\not<$ (*offset* $+$ LR-NODE-SIZE)))
$\rightarrow$ lr-proper-heapp-nodep (tag ('addr, cons (identity (LR-HEAP-NAME), *offset*)),
lr-init-data-seg (*heap-size*))

THEOREM: lr-proper-heapp2-lr-init-data-seg-helper
((*heap-size* ≮ 2)
 ∧ (*offset* < length (cdr (assoc (LR-HEAP-NAME,
                                   lr-init-data-seg (*heap-size*)))))
 ∧ (*offset* ∈ **N**)
 ∧ lr-boundary-nodep (tag ('addr, cons (LR-HEAP-NAME, *offset*))))
→ lr-proper-heapp2 (tag ('addr, cons (LR-HEAP-NAME, *offset*)),
                    lr-init-data-seg (*heap-size*))

THEOREM: lr-proper-heapp2-lr-init-data-seg
(*heap-size* ≮ 2)
→ lr-proper-heapp2 (tag ('addr,
                         cons (identity (LR-HEAP-NAME),
                               identity (LR-NODE-SIZE) ∗ *heap-size*)),
                    lr-init-data-seg (*heap-size*))

THEOREM: fetch-add-addr-ref-count-lr-t-addr-lr-init-data-seg
(*heap-size* ≮ 4)
→ (fetch (identity (add-addr (LR-T-ADDR, LR-REF-COUNT-OFFSET)),
          lr-init-data-seg (*heap-size*))
   = identity (add-addr (LR-T-ADDR, LR-NODE-SIZE)))

THEOREM: fetch-t-addr-compile-quote-list-init-data-seg
((*heap-size* ≮ 4) ∧ (¬ definedp (**t**, *table*)))
→ (fetch (identity (LR-T-ADDR),
          car (lr-compile-quote ('list,
                                 list (**t**, 0),
                                 lr-init-data-seg (*heap-size*),
                                 table)))
   = identity (tag ('nat, LR-TRUE-TAG)))

THEOREM: fetch-ref-count-t-addr-compile-quote-list-init-data-seg
((*heap-size* ≮ 4) ∧ (¬ definedp (**t**, *table*)))
→ (fetch (identity (add-addr (LR-T-ADDR, LR-REF-COUNT-OFFSET)),
          car (lr-compile-quote ('list,
                                 list (**t**, 0),
                                 lr-init-data-seg (*heap-size*),
                                 table)))
   = identity (tag ('nat, 1)))

THEOREM: fetch-add-addr-ref-count-lr-0-addr-lr-init-data-seg
(*heap-size* ≮ 4)
→ (fetch (identity (add-addr (LR-0-ADDR, LR-REF-COUNT-OFFSET)),
          lr-init-data-seg (*heap-size*))
   = identity (add-addr (LR-0-ADDR, LR-NODE-SIZE)))

THEOREM: fetch-0-addr-compile-quote-list-init-data-seg
((*heap-size* ≮ 4) ∧ (¬ definedp (**t**, *table*)) ∧ (¬ definedp (0, *table*)))
→ (fetch (identity (LR-0-ADDR),
car (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
table)))
= identity (tag ('nat, LR-ADD1-TAG)))

THEOREM: fetch-ref-count-0-addr-compile-quote-list-init-data-seg
((*heap-size* ≮ 4) ∧ (¬ definedp (**t**, *table*)) ∧ (¬ definedp (0, *table*)))
→ (fetch (identity (add-addr (LR-0-ADDR, LR-REF-COUNT-OFFSET)),
car (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
table)))
= identity (tag ('nat, 1)))

THEOREM: fetch-unbox-nat-0-addr-compile-quote-list-init-data-seg
((*heap-size* ≮ 4) ∧ (¬ definedp (**t**, *table*)) ∧ (¬ definedp (0, *table*)))
→ (fetch (identity (add-addr (LR-0-ADDR, LR-UNBOX-NAT-OFFSET)),
car (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
table)))
= identity (tag ('nat, 0)))

THEOREM: lr-proper-heapp-lr-compile-quote-ft-lr-init-data-seg
(*heap-size* ≮ 4)
→ lr-proper-heapp (car (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
list (cons (**f**, identity (LR-F-ADDR))))))

THEOREM: cdr-compile-quote-list-t0-lr-init-data-seg-cons-table
((¬ definedp (0, *table*)) ∧ (¬ definedp (**t**, *table*)) ∧ (*heap-size* ≮ 4))
→ (cdr (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
table))
= cons (cons (0, identity (LR-0-ADDR)),
cons (cons (**t**, identity (LR-T-ADDR)), *table*)))

THEOREM: fetch-f-addr-compile-quote-list-init-data-seg
((*heap-size* ≮ 4) ∧ (¬ definedp (**t**, *table*)) ∧ (¬ definedp (0, *table*)))

→ (fetch (identity (LR-F-ADDR),
car (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
table)))
= identity (tag ('nat, LR-FALSE-TAG)))

THEOREM: fetch-ref-count-f-addr-compile-quote-list-init-data-seg
((*heap-size* ≮ 4) ∧ (¬ definedp (**t**, *table*)) ∧ (¬ definedp (0, *table*)))
→ (fetch (identity (add-addr (LR-F-ADDR, LR-REF-COUNT-OFFSET)),
car (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
table)))
= identity (tag ('nat, 1)))

THEOREM: lr-s-similar-const-table-compile-quote-t0-init-data-seg
(*heap-size* ≮ 4)
→ lr-s-similar-const-table (cdr (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
list (cons (**f**,
identity (LR-F-ADDR))))),
car (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
list (cons (**f**,
identity (LR-F-ADDR))))))

EVENT: Disable cdr-compile-quote-list-t0-lr-init-data-seg-cons-table.

THEOREM: fetch-fp-addr-compile-quote-list-t0-lr-init-data-seg-cons-table
((¬ definedp (0, *table*)) ∧ (¬ definedp (**t**, *table*)) ∧ (*heap-size* ≮ 4))
→ (fetch (identity (LR-FP-ADDR),
car (lr-compile-quote ('list,
list (**t**, 0),
lr-init-data-seg (*heap-size*),
table)))
= identity (add-addr (LR-0-ADDR, LR-NODE-SIZE)))

THEOREM: lr-free-list-nodes-lr-compile-quote-t0
((¬ definedp (**t**, *table*)) ∧ (¬ definedp (0, *table*)) ∧ (*heap-size* ≮ 4))
→ (lr-free-list-nodes (tag ('addr,
cons (identity (LR-HEAP-NAME),

identity (LR-NODE-SIZE) ∗ *heap-size*)),
car (lr-compile-quote (`'list`,
list (**t**, `0`),
lr-init-data-seg (*heap-size*),
table)))
= lr-all-nodes (identity (LR-NODE-SIZE + offset (LR-0-ADDR)),
tag (`'addr`,
cons (identity (LR-HEAP-NAME),
identity (LR-NODE-SIZE) ∗ *heap-size*))))

THEOREM: listp-lr-all-nodes
listp (lr-all-nodes (*min-offset*, *max-addr*))
= ((offset (*max-addr*) ≠ `0`) ∧ (*min-offset* < offset (*max-addr*)))

THEOREM: length-lr-all-nodes
(lr-boundary-offsetp (*offset*) ∧ (*offset* ∈ **N**) ∧ lr-boundary-nodep (*addr*))
→ (length (lr-all-nodes (*offset*, *addr*))
= ((offset (*addr*) − *offset*) ÷ identity (LR-NODE-SIZE)))

THEOREM: lr-count-free-nodes-append-lr-all-nodes-fact
(((LR-NODE-SIZE ∗ *heap-size*) ≮ LR-MINIMUM-HEAP-SIZE)
∧ (¬ definedp (**t**, *table*))
∧ (¬ definedp (`0`, *table*)))
→ (lr-count-free-nodes (identity (add-addr (LR-0-ADDR, LR-NODE-SIZE)),
lr-all-nodes (identity (LR-MINIMUM-HEAP-SIZE),
tag (`'addr`,
cons (identity (LR-HEAP-NAME),
identity (LR-NODE-SIZE)
∗ *heap-size*))),
car (lr-compile-quote (`'list`,
list (**t**, `0`),
lr-init-data-seg (*heap-size*),
table)))
= (*heap-size* − LR-NODE-SIZE))

THEOREM: proper-p-alistp-pair-formal-with-addresses
(all-litatoms (strip-cars (*params*))
∧ s-init-data-seg-restrictedp (*params*)
∧ (*heap-size* ≮ s-total-heap-reqs (s-progs (*s*), *params*, *heap-size*))
∧ (*word-size* ≮ s-total-ws-reqs (s-progs (*s*), *params*, *heap-size*)))
→ proper-p-alistp (pair-formals-with-addresses (*params*,
cdr (lr-data-seg-table (s-progs (*s*),
params,
heap-size))),
lr->p (s->lr1 (*s*,

p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
car (lr-data-seg-table (s-progs (*s*),
params,
heap-size)),
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw),
cdr (lr-data-seg-table (s-progs (*s*),
params,
heap-size)))))

THEOREM: proper-p-alistp-lr-make-initial-temps
(lr-proper-heapp (*data-seg*)
∧ all-litatoms (*temp-vars*)
∧ lr-proper-p-areasp (*data-seg*))
→ proper-p-alistp (lr-make-initial-temps (*temp-vars*),
lr->p (s->lr1 (*s*,
p-state (*pc*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl,
max-temp,
word-size,
psw),
table)))

THEOREM: lr-minimum-heapp-lr-data-seg-table-body
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (lr-minimum-heapp (car (lr-data-seg-table-body (*flag*, *body*, *data-seg*, *table*)))
= lr-minimum-heapp (*data-seg*))

THEOREM: lr-count-free-nodes-lr-compile-quote-s-heap-reqs-flag-t
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-minimum-heapp (*data-seg*)

$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (**t**, *object*, *data-seg*, *table*)))
$\rightarrow$ ((s-heap-reqs (**t**, *object*, *data-seg*, *table*)
$+$ lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
car (lr-compile-quote (**t**,
object,
data-seg,
table))),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (lr-compile-quote (**t**,
object,
data-seg,
table))),
car (lr-compile-quote (**t**,
object,
data-seg,
table))))
$=$ lr-count-free-nodes (fetch (identity (LR-FP-ADDR), *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg))

THEOREM: lr-count-free-nodes-lr-data-seg-table-body-s-heap-reqs
(lr-proper-free-listp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-body (*flag*, *body*, *data-seg*, *table*)))
$\rightarrow$ ((lr-count-free-nodes (fetch (LR-FP-ADDR,
car (lr-data-seg-table-body (*flag*,
body,
data-seg,
table))),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (lr-data-seg-table-body (*flag*,

body,
data-seg,
table))),
car (lr-data-seg-table-body (*flag*,
body,
data-seg,
table)))
\+ s-heap-reqs-body (*flag*, *body*, *data-seg*, *table*))
= lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg))

THEOREM: lr-data-seg-table-body-lr-good-pointerp-tablep-help1
let *data-seg-tab* **be** lr-data-seg-table-body (**t**, car (*body*), *data-seg*, *table*)
in
(listp (*body*)
∧ lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-minimum-heapp (*data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*))
∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ (s-heap-reqs-body (**t**, car (*body*), *data-seg*, *table*)
\+ s-heap-reqs-body (`'list`,
cdr (*body*),
car (*data-seg-tab*),
cdr (*data-seg-tab*)))))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
car (*data-seg-tab*)),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (*data-seg-tab*)),
car (*data-seg-tab*))
$\not<$ s-heap-reqs-body (`'list`,
cdr (*body*),
car (*data-seg-tab*),
cdr (*data-seg-tab*))) **endlet**

THEOREM: lr-data-seg-table-body-lr-good-pointerp-tablep
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)

$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-data-seg-body-restrictedp (*flag*, *body*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-body (*flag*, *body*, *data-seg*, *table*)))
$\rightarrow$ lr-good-pointerp-tablep (cdr (lr-data-seg-table-body (*flag*,
body,
data-seg,
table)),
car (lr-data-seg-table-body (*flag*,
body,
data-seg,
table)))

THEOREM: lr-data-seg-table-list-lr-good-pointerp-tablep-helper-1
let *dst-body* **be** lr-data-seg-table-body (**t**,
s-body (car (*progs*)),
data-seg,
table)
in
(listp (*progs*)
$\wedge$ lr-proper-free-listp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ (s-heap-reqs-body (**t**, s-body (car (*progs*)), *data-seg*, *table*)
$+$ s-heap-reqs-list (cdr (*progs*),
car (*dst-body*),
cdr (*dst-body*)))))
$\rightarrow$ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR), car (*dst-body*)),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (*dst-body*)),
car (*dst-body*))

$\not<$ s-heap-reqs-list (cdr (*progs*),
car (*dst-body*),
cdr (*dst-body*))) **endlet**

THEOREM: lr-init-data-seg-table-lr-good-pointerp-tablep
(lr-proper-free-listp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-init-data-seg-restrictedp (*params*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-init-heap-reqs (*params*, *data-seg*, *table*)))
$\rightarrow$ lr-good-pointerp-tablep (cdr (lr-init-data-seg-table (*params*,
data-seg,
table)),
car (lr-init-data-seg-table (*params*,
data-seg,
table)))

THEOREM: lr-proper-heapp-car-lr-data-seg-table-body
(lr-proper-heapp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-data-seg-body-restrictedp (*flag*, *body*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-body (*flag*, *body*, *data-seg*, *table*)))
$\rightarrow$ lr-proper-heapp (car (lr-data-seg-table-body (*flag*, *body*, *data-seg*, *table*)))

THEOREM: lr-proper-heapp-car-lr-data-seg-table-list
(lr-proper-heapp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-data-seg-list-restrictedp (*progs*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),

lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-list (*progs*, *data-seg*, *table*)))
$\rightarrow$ lr-proper-heapp (car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))

THEOREM: lr-proper-heapp-car-lr-init-data-seg-table
(lr-proper-heapp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-init-data-seg-restrictedp (*params*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-init-heap-reqs (*params*, *data-seg*, *table*)))
$\rightarrow$ lr-proper-heapp (car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))

THEOREM: lr-count-free-nodes-lr-init-data-seg-table-s-init-heap-reqs
(lr-proper-free-listp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-init-heap-reqs (*params*, *data-seg*, *table*)))
$\rightarrow$ ((lr-count-free-nodes (fetch (LR-FP-ADDR,
car (lr-init-data-seg-table (*params*,
data-seg,
table))),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (lr-init-data-seg-table (*params*,
data-seg,
table))),
car (lr-init-data-seg-table (*params*,
data-seg,
table)))
$+$ s-init-heap-reqs (*params*, *data-seg*, *table*))
$=$ lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),

data-seg),
data-seg))

THEOREM: lr-proper-heapp-car-lr-data-seg-table-helper-1
let *data-seg-table* **be** lr-compile-quote (`'list`,
list (**t**, `0`),
lr-init-data-seg (*heap-size*),
list (cons (**f**, identity (LR-F-ADDR))))
in
((*heap-size* ≮ (`2`
\+ `2`
\+ s-init-heap-reqs (*params*,
car (*data-seg-table*),
cdr (*data-seg-table*))
\+ *x*))
∧ (*max-addr* = lr-max-node (car (*data-seg-table*))))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
car (lr-init-data-seg-table (*params*,
car (*data-seg-table*),
cdr (*data-seg-table*)))),
lr-free-list-nodes (*max-addr*,
car (lr-init-data-seg-table (*params*,
car (*data-seg-table*),
cdr (*data-seg-table*)))),
car (lr-init-data-seg-table (*params*,
car (*data-seg-table*),
cdr (*data-seg-table*))))
≮ *x*) **endlet**

THEOREM: lr-proper-heapp-car-lr-data-seg-table
((*heap-size* ≮ s-total-heap-reqs (*progs*, *params*, *heap-size*))
∧ s-restrictedp (*progs*, *params*))
→ lr-proper-heapp (car (lr-data-seg-table (*progs*, *params*, *heap-size*)))

THEOREM: litatom-car-gensym
litatom (car (gensym (*initial*, *num-list*, *atom-list*)))

THEOREM: all-litatoms-strip-cdrs-lr-make-temp-name-alist-1
all-litatoms (strip-cdrs (lr-make-temp-name-alist-1 (*initial*,
num-list,
temp-list,
formals)))

THEOREM: all-litatoms-strip-cdrs-lr-make-temp-name-alist
all-litatoms (strip-cdrs (lr-make-temp-name-alist (*temp-list*, *formals*)))

THEOREM: lr-proper-p-areasp-car-lr-data-seg-table
lr-proper-p-areasp (car (lr-data-seg-table (*progs*, *params*, *heap-size*)))

THEOREM: plist-strip-cdrs
plist (strip-cdrs (*x*)) = strip-cdrs (*x*)

THEOREM: lr-make-temp-name-alist-1-plist-arg-1
lr-make-temp-name-alist-1 (*initial*, *num-list*, plist (*temp-list*), *formals*)
= lr-make-temp-name-alist-1 (*initial*, *num-list*, *temp-list*, *formals*)

THEOREM: lr-make-temp-name-alist-plist-arg-1
lr-make-temp-name-alist (plist (*temp-list*), *formals*)
= lr-make-temp-name-alist (*temp-list*, *formals*)

THEOREM: length-lr-make-initial-temps
length (lr-make-initial-temps (*temp-vars*)) = length (*temp-vars*)

THEOREM: length-strip-cdrs
length (strip-cdrs (*alist*)) = length (*alist*)

THEOREM: length-pair-formals-with-addresses
length (pair-formals-with-addresses (*formals*, *alist*)) = length (*formals*)

THEOREM: s-good-statep-length-s-temp-list
s-good-statep (*s*, *c*)
→ (length (s-temp-list (s-prog (*s*))) = length (s-temps (*s*)))

THEOREM: plistp-comp-programs-1
plistp (comp-programs-1 (*progs*))

THEOREM: proper-p-instructionp-ret
proper-p-instructionp (`'(ret)`, *name*, *p*)

THEOREM: proper-p-instructionp-eq
proper-p-instructionp (`'(eq)`, *name*, *p*)

THEOREM: proper-p-instructionp-fetch
proper-p-instructionp (`'(fetch)`, *name*, *p*)

THEOREM: proper-p-instructionp-deposit
proper-p-instructionp (`'(deposit)`, *name*, *p*)

THEOREM: proper-p-instructionp-add-addr
proper-p-instructionp (`'(add-addr)`, *name*, *p*)

THEOREM: proper-p-instructionp-pop-global-free-ptr
lr-proper-heapp (p-data-segment (*l*))
→ proper-p-instructionp (`'(pop-global free-ptr)`, *name*, lr->p (*l*))

THEOREM: proper-p-instructionp-push-global-free-ptr
lr-proper-heapp (p-data-segment (*l*))
→ proper-p-instructionp (`'(push-global free-ptr)`, *name*, lr->p (*l*))

THEOREM: proper-p-instructionp-push-local-temp-car
lr-programs-properp (*l*, *table*)
→ proper-p-instructionp (`'(push-local x)`, `'car`, lr->p (*l*))

THEOREM: proper-p-instructionp-push-local-temp-cdr
lr-programs-properp (*l*, *table*)
→ proper-p-instructionp (`'(push-local x)`, `'cdr`, lr->p (*l*))

THEOREM: proper-p-instructionp-push-local-temp-cons
lr-programs-properp (*l*, *table*)
→ proper-p-instructionp (`'(push-local temp)`, `'cons`, lr->p (*l*))

THEOREM: proper-p-instructionp-set-local-temp-cons
lr-programs-properp (*l*, *table*)
→ proper-p-instructionp (`'(set-local temp)`, `'cons`, lr->p (*l*))

THEOREM: proper-labeled-p-instructionsp-append
plistp (*instrs1*)
→ (proper-labeled-p-instructionsp (append (*instrs1*, *instrs2*), *name*, *p*)
= (proper-labeled-p-instructionsp (*instrs1*, *name*, *p*)
∧ proper-labeled-p-instructionsp (*instrs2*, *name*, *p*)))

THEOREM: lessp-number-cons-cur-expr-dv-1-listp
listp (cur-expr (*pos*, *body*))
→ (number-cons (cur-expr (dv (*pos*, `1`), *body*))
< number-cons (cur-expr (*pos*, *body*)))

THEOREM: lessp-number-cons-cur-expr-dv-2-listp
listp (cur-expr (*pos*, *body*))
→ (number-cons (cur-expr (dv (*pos*, `2`), *body*))
< number-cons (cur-expr (*pos*, *body*)))

THEOREM: lessp-number-cons-cur-expr-dv-3-listp
listp (cur-expr (*pos*, *body*))
→ (number-cons (cur-expr (dv (*pos*, `3`), *body*))
< number-cons (cur-expr (*pos*, *body*)))

THEOREM: lessp-number-cons-restn-cdr
(listp (*pos*)
∧ (car (last (*pos*)) < length (cur-expr (butlast (*pos*), *body*)))
∧ listp (cur-expr (butlast (*pos*), *body*)))
→ (number-cons (restn (car (last (*pos*)), cdr (cur-expr (butlast (*pos*), *body*))))
< number-cons (restn (car (last (*pos*)), cur-expr (butlast (*pos*), *body*))))

THEOREM: proper-p-instructionp-test-bool-and-jump-label
proper-p-instructionp (list ('test-bool-and-jump, x, lab), $name$, p)
= find-labelp (lab, program-body (assoc ($name$, p-prog-segment (p))))

THEOREM: proper-p-instructionp-jump-label
proper-p-instructionp (list ('jump, lab), $name$, p)
= find-labelp (lab, program-body (assoc ($name$, p-prog-segment (p))))

THEOREM: plist-lr-convert-num-to-ascii
plistp (lr-convert-num-to-ascii ($number$, $list$)) = plistp ($list$)

THEOREM: zerop-lr-convert-num-to-ascii
($number \simeq 0$)
→ (lr-convert-num-to-ascii ($number$, $list$) = cons (ASCII-0, $list$))

THEOREM: lr-convert-digit-to-ascii-equal
(lr-convert-digit-to-ascii (m) = lr-convert-digit-to-ascii (n))
= (fix (m) = fix (n))

THEOREM: zerop-lr-convert-digit-to-ascii
($number \simeq 0$) → (lr-convert-digit-to-ascii ($number$) = ASCII-0)

THEOREM: equal-ascii-0-lr-convert-digit-to-ascii
(lr-convert-digit-to-ascii ($number$) = identity (ASCII-0)) = ($number \simeq 0$)

THEOREM: length-lr-convert-num-to-ascii
length (lr-convert-num-to-ascii ($number$, $list$)) $\not<$ (1 + length ($list$))

DEFINITION:
induct-hint-22 (n, m, $list$)
= **if** $n < 10$ **then t**
elseif $m < 10$ **then t**
else induct-hint-22 ($n \div 10$,
$m \div 10$,
cons (lr-convert-digit-to-ascii (n **mod** 10),
$list$)) **endif**

THEOREM: lr-convert-num-to-ascii-equal-arg1
(lr-convert-num-to-ascii (x, $list1$) = lr-convert-num-to-ascii (x, $list2$))
= ($list1$ = $list2$)

DEFINITION:
induct-hint-23 (n, m, $list1$, $list2$)
= **if** $n < 10$ **then t**
elseif $m < 10$ **then t**
else induct-hint-23 ($n \div 10$,

$m \div 10$,
cons (lr-convert-digit-to-ascii (n **mod** 10),
list1),
cons (lr-convert-digit-to-ascii (m **mod** 10),
list2)) **endif**

THEOREM: lr-convert-num-to-ascii-equal-arg2-lengths-helper-1
(length (*list1*) = length (*list2*))
→ (lr-convert-num-to-ascii (*number*, cons (x, *list1*)) ≠ cons (y, *list2*))

THEOREM: lr-convert-num-to-ascii-equal-arg2-lengths
((length (*list1*) = length (*list2*)) ∧ (*list1* ≠ *list2*))
→ (lr-convert-num-to-ascii (x, *list1*)
≠ lr-convert-num-to-ascii (y, *list2*))

THEOREM: lr-convert-num-to-ascii-equal-arg2-helper-1
($x \neq y$)
→ (lr-convert-num-to-ascii (w, cons (x, *list*))
≠ lr-convert-num-to-ascii (z, cons (y, *list*)))

THEOREM: lr-convert-num-to-ascii-equal-arg2
(lr-convert-num-to-ascii (m, *list*) = lr-convert-num-to-ascii (n, *list*))
= (fix (m) = fix (n))

THEOREM: lr-make-label-equal
(lr-make-label (m) = lr-make-label (n)) = (fix (m) = fix (n))

THEOREM: find-labelp-lr-make-label-label-instrs
find-labelp (lr-make-label (m), label-instrs (*instrs*, n))
= (($m \not< n$) ∧ ($m <$ (n + length (*instrs*))))

THEOREM: find-labelp-lr-make-label-comp-body
($n <$ length (comp-body (*body*)))
→ find-labelp (lr-make-label (n), comp-body (*body*))

THEOREM: label-instrs-append
($n \in \mathbf{N}$)
→ (label-instrs (append (*instrs1*, *instrs2*), n)
= append (label-instrs (*instrs1*, n),
label-instrs (*instrs2*, n + length (*instrs1*))))

THEOREM: proper-p-temp-var-dclsp-all-litatoms-all-undef-addrs
(all-litatoms (strip-cars (*temp-var-dcls*))
∧ all-undef-addrs (strip-cadrs (*temp-var-dcls*))
∧ lr-proper-heapp (p-data-segment (p))
∧ lr-proper-p-areasp (p-data-segment (p)))
→ proper-p-temp-var-dclsp (*temp-var-dcls*, p)

THEOREM: plistp-label-instrs
plistp (label-instrs (*instrs*, *n*))

DEFINITION:
not-labelledp-instrs (*instrs*)
= **if** listp (*instrs*)
then (¬ labelledp (car (*instrs*)))
∧ not-labelledp-instrs (cdr (*instrs*))
else t endif

THEOREM: label-instrs-proper-labeled-p-instructionsp
(proper-labeled-p-instructionsp (*instrs*, *name*, *p*)
∧ not-labelledp-instrs (*instrs*))
→ proper-labeled-p-instructionsp (label-instrs (*instrs*, *n*), *name*, *p*)

THEOREM: not-labelledp-instrs-append
not-labelledp-instrs (append (*instrs1*, *instrs2*))
= (not-labelledp-instrs (*instrs1*) ∧ not-labelledp-instrs (*instrs2*))

THEOREM: not-labelledp-instrs-comp-body-1
not-labelledp-instrs (comp-body-1 (*flag*, *body*, *n*))

THEOREM: comp-body-1-list-not-listp
(¬ listp (*body*)) → (comp-body-1 (`'list`, *body*, *n*) = **nil**)

THEOREM: proper-labeled-p-instructionsp-nil
proper-labeled-p-instructionsp (**nil**, *name*, *p*)

THEOREM: proper-labeled-p-instructionsp-comp-body-1-helper-1
(proper-labeled-p-instructionsp (comp-body-1 (**t**, *test-body*, *n1*),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))
∧ proper-labeled-p-instructionsp (comp-body-1 (**t**, *then-body*, *n2*),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,

comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

∧ proper-labeled-p-instructionsp (*else-instrs*,
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

∧ lr-proper-heapp (*data-seg*)

∧ lr-proper-p-areasp (*data-seg*)

∧ definedp (*name*, *prog-seg*)

∧ (((*n*
\+ lr-p-c-size (**t**, *test-body*)
\+ lr-p-c-size (**t**, *then-body*)
\+ length (*else-instrs*)
\+ 4)− 1)
< lr-p-c-size (**t**, program-body (assoc (*name*, *prog-seg*)))))

→ proper-labeled-p-instructionsp (comp-if (comp-body-1 (**t**, *test-body*, *n1*),
comp-body-1 (**t**, *then-body*, *n2*),
else-instrs,
n),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

THEOREM: proper-labeled-p-instructionsp-comp-body-1-helper-2
(listp (*body*)
∧ ((car (*body*) = S-TEMP-FETCH) ∨ (car (*body*) = S-TEMP-TEST))

$\wedge$ lr-proper-exprp (**t**,
body,
program-names,
formal-vars (assoc (*name*, *prog-seg*)),
strip-cars (temp-var-dcls (assoc (*name*, *prog-seg*))),
table)
$\wedge$ definedp (*name*, *prog-seg*))
$\rightarrow$ proper-labeled-p-instructionsp (list (list (`'push-local`, caddr (*body*))),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

THEOREM: proper-labeled-p-instructionsp-comp-body-1-helper-3
(listp (*body*)
$\wedge$ ((car (*body*) = S-TEMP-EVAL) $\vee$ (car (*body*) = S-TEMP-TEST))
$\wedge$ lr-proper-exprp (**t**,
body,
program-names,
formal-vars (assoc (*name*, *prog-seg*)),
strip-cars (temp-var-dcls (assoc (*name*, *prog-seg*))),
table)
$\wedge$ definedp (*name*, *prog-seg*))
$\rightarrow$ proper-labeled-p-instructionsp (list (list (`'set-local`, caddr (*body*))),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

THEOREM: proper-labeled-p-instructionsp-comp-body-1-helper-4
(listp (*body*)
$\wedge$ (car (*body*) = S-TEMP-TEST)

$$
\begin{array}{ll}
\wedge & \text{proper-labeled-p-instructionsp}\,(\text{comp-body-1}\,(\mathbf{t},\ \text{cadr}\,(\mathit{body}),\ n+4),\\
& \qquad \mathit{name},\\
& \qquad \text{p-state}\,(\mathit{pc},\\
& \qquad\qquad \mathit{ctrl\text{-}stk},\\
& \qquad\qquad \mathit{temp\text{-}stk},\\
& \qquad\qquad \text{comp-programs}\,(\mathit{prog\text{-}seg}),\\
& \qquad\qquad \mathit{data\text{-}seg},\\
& \qquad\qquad \mathit{max\text{-}ctrl},\\
& \qquad\qquad \mathit{max\text{-}temp},\\
& \qquad\qquad \mathit{word\text{-}size},\\
& \qquad\qquad \mathit{psw}))\\
\wedge & \text{lr-proper-exprp}\,(\mathbf{t},\\
& \qquad \mathit{body},\\
& \qquad \mathit{program\text{-}names},\\
& \qquad \text{formal-vars}\,(\text{assoc}\,(\mathit{name},\ \mathit{prog\text{-}seg})),\\
& \qquad \text{strip-cars}\,(\text{temp-var-dcls}\,(\text{assoc}\,(\mathit{name},\ \mathit{prog\text{-}seg}))),\\
& \qquad \mathit{table})\\
\wedge & \text{lr-proper-heapp}\,(\mathit{data\text{-}seg})\\
\wedge & \text{lr-proper-p-areasp}\,(\mathit{data\text{-}seg})\\
\wedge & (\text{lr-p-c-size}\,(\mathbf{t},\ \text{program-body}\,(\text{assoc}\,(\mathit{name},\ \mathit{prog\text{-}seg})))\\
& \quad \not< \quad (n + \text{lr-p-c-size}\,(\mathbf{t},\ \text{cadr}\,(\mathit{body})) + 7))\\
\wedge & \text{definedp}\,(\mathit{name},\ \mathit{prog\text{-}seg}))\\
\rightarrow & \text{proper-labeled-p-instructionsp}\,(\text{comp-temp-test}\,(\mathit{body},\\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{comp-body-1}\,(\mathbf{t},\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{cadr}\,(\mathit{body}),\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad n+4),\\
& \qquad\qquad\qquad\qquad\qquad\qquad n),\\
& \qquad \mathit{name},\\
& \qquad \text{p-state}\,(\mathit{pc},\\
& \qquad\qquad \mathit{ctrl\text{-}stk},\\
& \qquad\qquad \mathit{temp\text{-}stk},\\
& \qquad\qquad \text{comp-programs}\,(\mathit{prog\text{-}seg}),\\
& \qquad\qquad \mathit{data\text{-}seg},\\
& \qquad\qquad \mathit{max\text{-}ctrl},\\
& \qquad\qquad \mathit{max\text{-}temp},\\
& \qquad\qquad \mathit{word\text{-}size},\\
& \qquad\qquad \mathit{psw}))
\end{array}
$$

DEFINITION:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{assoc-cdr}\,(x,\ l)}\\
= & \textbf{if}\ \text{listp}\,(l)\\
& \textbf{then if}\ x = \text{cdar}\,(l)\ \ \textbf{then}\ \text{car}\,(l)\\
& \qquad \textbf{else}\ \text{assoc-cdr}\,(x,\ \text{cdr}\,(l))\ \textbf{endif}\\
& \textbf{else f endif}
\end{array}
$$

THEOREM: lr-s-similar-const-table-lr-valp-member-strip-cdrs
((*addr* ∈ strip-cdrs (*table*)) ∧ lr-s-similar-const-table (*table*, *data-seg*))
→ lr-valp (car (assoc-cdr (*addr*, *table*)), *addr*, *data-seg*)

THEOREM: lr-s-similar-const-table-type-addr-member-strip-cdrs
((*addr* ∈ strip-cdrs (*table*)) ∧ lr-s-similar-const-table (*table*, *data-seg*))
→ ((type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ adpp (untag (*addr*), *data-seg*)
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = identity (LR-HEAP-NAME))
∧ (type (fetch (add-addr (*addr*, identity (LR-REF-COUNT-OFFSET)),
data-seg))
= 'nat))

THEOREM: proper-labeled-p-instructionsp-comp-body-1-helper-5
(listp (*body*)
∧ (car (*body*) = 'quote)
∧ lr-proper-heapp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ lr-proper-exprp (**t**,
body,
strip-logic-fnames (cdr (*prog-seg*)),
formal-vars (assoc (*name*, *prog-seg*)),
strip-cars (temp-var-dcls (assoc (*name*, *prog-seg*))),
table)
∧ lr-s-similar-const-table (*table*, *data-seg*)
∧ definedp (*name*, *prog-seg*))
→ proper-labeled-p-instructionsp (list (list ('push-constant,
cadr (*body*))),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

EVENT: Disable lr-s-similar-const-table-type-addr-member-strip-cdrs.

THEOREM: lr-proper-exprp-flag-list-cdr-funcall

(listp (*body*)
∧ (car (*body*) ≠ 'if)
∧ (car (*body*) ≠ S-TEMP-FETCH)
∧ (car (*body*) ≠ S-TEMP-EVAL)
∧ (car (*body*) ≠ S-TEMP-TEST)
∧ (car (*body*) ≠ 'quote)
∧ lr-proper-exprp (**t**, *body*, *program-names*, *formals*, *temps*, *table*))
→ lr-proper-exprp ('list, cdr (*body*), *program-names*, *formals*, *temps*, *table*)

THEOREM: proper-labeled-p-instructionsp-comp-body-1-helper-6-1
(listp (*body*)
∧ (car (*body*) ≠ 'if)
∧ (car (*body*) ≠ S-TEMP-FETCH)
∧ (car (*body*) ≠ S-TEMP-EVAL)
∧ (car (*body*) ≠ S-TEMP-TEST)
∧ (car (*body*) ≠ 'quote)
∧ lr-proper-exprp (**t**,
body,
strip-logic-fnames (cdr (*prog-seg*)),
formals,
temps,
table)
∧ definedp (car (*body*), P-RUNTIME-SUPPORT-PROGRAMS))
→ proper-labeled-p-instructionsp (list (list ('call, car (*body*))),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

THEOREM: proper-labeled-p-instructionsp-comp-body-1-helper-6-2
(listp (*body*)
∧ (car (*body*) ≠ 'if)
∧ (car (*body*) ≠ S-TEMP-FETCH)
∧ (car (*body*) ≠ S-TEMP-EVAL)
∧ (car (*body*) ≠ S-TEMP-TEST)
∧ (car (*body*) ≠ 'quote)
∧ lr-proper-exprp (**t**,
body,

strip-logic-fnames (cdr (*prog-seg*)),
formals,
temps,
table)
∧ (¬ definedp (car (*body*), P-RUNTIME-SUPPORT-PROGRAMS))
∧ all-user-fnamesp (cdr (strip-cars (*prog-seg*)))
∧ definedp (*name*, *prog-seg*))
→ proper-labeled-p-instructionsp (list (list (`'call`,
user-fname (car (*body*)))),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

THEOREM: proper-labeled-p-instructionsp-comp-body-1-helper-7
((¬ listp (*body*))
∧ lr-proper-exprp (**t**,
body,
program-names,
formal-vars (assoc (*name*, *prog-seg*)),
temps,
table)
∧ definedp (*name*, *prog-seg*))
→ proper-labeled-p-instructionsp (list (list (`'push-local`, *body*)),
name,
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

THEOREM: not-definedp-not-listp
(¬ listp (*alist*)) → (¬ definedp (*name*, *alist*))

THEOREM: proper-labeled-p-instructionsp-comp-body-1

(lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ all-user-fnamesp (cdr (strip-cars (*prog-seg*)))
 ∧ lr-proper-exprp (**if** *flag* = `'list` **then** `'list`
 else t endif,
 body,
 strip-logic-fnames (cdr (*prog-seg*)),
 formal-vars (assoc (*name*, *prog-seg*)),
 strip-cars (temp-var-dcls (assoc (*name*, *prog-seg*))),
 table)
 ∧ lr-s-similar-const-table (*table*, *data-seg*)
 ∧ ((*n* + lr-p-c-size (**if** *flag* = `'list` **then** `'list`
 else t endif,
 body))
 < (1 + lr-p-c-size (**t**, program-body (assoc (*name*, *prog-seg*)))))
 ∧ definedp (*name*, *prog-seg*))
→ proper-labeled-p-instructionsp (comp-body-1 (*flag*, *body*, *n*),
 name,
 p-state (*pc*,
 ctrl-stk,
 temp-stk,
 comp-programs (*prog-seg*),
 data-seg,
 max-ctrl,
 max-temp,
 word-size,
 psw))

THEOREM: all-undef-addrs-strip-cadrs-lr-make-temp-var-dcls
all-undef-addrs (strip-cadrs (lr-make-temp-var-dcls (*temp-alist*)))

THEOREM: proper-p-programp-p-car-code
(lr-programs-properp (*l*, *table*)
 ∧ (p-word-size (*l*) ∈ **N**)
 ∧ (p-word-size (*l*) ≮ S-MAX-SUBR-REQS)
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ lr-proper-p-areasp (p-data-segment (*l*)))
→ proper-p-programp (P-CAR-CODE, lr->p (*l*))

THEOREM: proper-p-programp-p-cdr-code
(lr-programs-properp (*l*, *table*)
 ∧ (p-word-size (*l*) ∈ **N**)
 ∧ (p-word-size (*l*) ≮ S-MAX-SUBR-REQS)
 ∧ lr-proper-heapp (p-data-segment (*l*))
 ∧ lr-proper-p-areasp (p-data-segment (*l*)))

→ proper-p-programp (P-CDR-CODE, lr->p (l))

THEOREM: proper-p-programp-p-cons-code
(lr-programs-properp (l, *table*)
∧ (p-word-size (l) ∈ **N**)
∧ (p-word-size (l) ≮ S-MAX-SUBR-REQS)
∧ lr-proper-heapp (p-data-segment (l))
∧ lr-proper-p-areasp (p-data-segment (l)))
→ proper-p-programp (P-CONS-CODE, lr->p (l))

THEOREM: proper-p-programp-p-false-code
(lr-programs-properp (l, *table*)
∧ (p-word-size (l) ∈ **N**)
∧ (p-word-size (l) ≮ log (2, LR-FALSE-TAG))
∧ lr-proper-heapp (p-data-segment (l))
∧ lr-proper-p-areasp (p-data-segment (l)))
→ proper-p-programp (P-FALSE-CODE, lr->p (l))

THEOREM: proper-p-programp-p-falsep-code
(lr-programs-properp (l, *table*)
∧ (p-word-size (l) ∈ **N**)
∧ (p-word-size (l) ≮ log (2, LR-FALSE-TAG))
∧ lr-proper-heapp (p-data-segment (l))
∧ lr-proper-p-areasp (p-data-segment (l)))
→ proper-p-programp (P-FALSEP-CODE, lr->p (l))

THEOREM: proper-p-programp-p-listp-code
(lr-programs-properp (l, *table*)
∧ (p-word-size (l) ∈ **N**)
∧ (p-word-size (l) ≮ S-MAX-SUBR-REQS)
∧ lr-proper-heapp (p-data-segment (l))
∧ lr-proper-p-areasp (p-data-segment (l)))
→ proper-p-programp (P-LISTP-CODE, lr->p (l))

THEOREM: proper-p-programp-p-nlistp-code
(lr-programs-properp (l, *table*)
∧ (p-word-size (l) ∈ **N**)
∧ (p-word-size (l) ≮ S-MAX-SUBR-REQS)
∧ lr-proper-heapp (p-data-segment (l))
∧ lr-proper-p-areasp (p-data-segment (l)))
→ proper-p-programp (P-NLISTP-CODE, lr->p (l))

THEOREM: proper-p-programp-p-true-code
(lr-programs-properp (l, *table*)
∧ (p-word-size (l) ∈ **N**)

$\wedge$ (p-word-size (l) $\not<$ log (2, LR-TRUE-TAG))
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ lr-proper-p-areasp (p-data-segment (l)))
$\rightarrow$ proper-p-programp (P-TRUE-CODE, lr->p (l))

THEOREM: proper-p-programp-p-truep-code
(lr-programs-properp (l, *table*)
$\wedge$ (p-word-size (l) $\in \mathbf{N}$)
$\wedge$ (p-word-size (l) $\not<$ log (2, LR-TRUE-TAG))
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ lr-proper-p-areasp (p-data-segment (l)))
$\rightarrow$ proper-p-programp (P-TRUEP-CODE, lr->p (l))

THEOREM: lr-s-similar-const-table-lr-data-seg-table-body
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ s-data-seg-body-restrictedp (*flag*, *body*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-body (*flag*, *body*, *data-seg*, *table*)))
$\rightarrow$ lr-s-similar-const-table (cdr (lr-data-seg-table-body (*flag*,
body,
data-seg,
table)),
car (lr-data-seg-table-body (*flag*,
body,
data-seg,
table)))

THEOREM: lr-s-similar-const-table-lr-data-seg-table-list
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ s-data-seg-list-restrictedp (*progs*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-list (*progs*, *data-seg*, *table*)))
$\rightarrow$ lr-s-similar-const-table (cdr (lr-data-seg-table-list (*progs*,

data-seg,
table)),
car (lr-data-seg-table-list (*progs*,
data-seg,
table)))

THEOREM: lr-s-similar-const-table-lr-init-data-seg-table
(lr-proper-p-areasp (*data-seg*)
∧ lr-proper-heapp (*data-seg*)
∧ lr-s-similar-const-table (*table*, *data-seg*)
∧ s-init-data-seg-restrictedp (*params*)
∧ definedp (**f**, *table*)
∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-init-heap-reqs (*params*, *data-seg*, *table*)))
→ lr-s-similar-const-table (cdr (lr-init-data-seg-table (*params*,
data-seg,
table)),
car (lr-init-data-seg-table (*params*,
data-seg,
table)))

THEOREM: lr-s-similar-const-table-cdr-car-lr-data-seg-table
((*heap-size* $\not<$ s-total-heap-reqs (*progs*, *params*, *heap-size*))
∧ s-restrictedp (*progs*, *params*))
→ lr-s-similar-const-table (cdr (lr-data-seg-table (*progs*, *params*, *heap-size*)),
car (lr-data-seg-table (*progs*, *params*, *heap-size*)))

DEFINITION:
induct-hint-3 (*flag*, *pos*, *prog*)
= **if** *flag* = `'list`
then if *pos* ≃ **nil then t**
elseif listp (restn (car (last (*pos*)),
cur-expr (butlast (*pos*), s-body (*prog*))))
then induct-hint-3 (**t**, *pos*, *prog*)
∧ induct-hint-3 (`'list`, nx (*pos*), *prog*)
else t endif
elseif listp (cur-expr (*pos*, s-body (*prog*)))
then if car (cur-expr (*pos*, s-body (*prog*))) = `'if`
then induct-hint-3 (**t**, dv (*pos*, `1`), *prog*)
∧ induct-hint-3 (**t**, dv (*pos*, `2`), *prog*)
∧ induct-hint-3 (**t**, dv (*pos*, `3`), *prog*)
elseif car (cur-expr (*pos*, s-body (*prog*))) = S-TEMP-FETCH **then t**

elseif (car (cur-expr (*pos*, s-body (*prog*))) = S-TEMP-EVAL)
∨ (car (cur-expr (*pos*, s-body (*prog*))) = S-TEMP-TEST)
then induct-hint-3 (**t**, dv (*pos*, `1`), *prog*)
elseif car (cur-expr (*pos*, s-body (*prog*))) = `'quote` **then t**
else induct-hint-3 (`'list`, dv (*pos*, `1`), *prog*) **endif**
else t endif

THEOREM: lr-proper-exprp-p-lr-compile-programs-helper-1
listp (*pos*)
→ (lr-compile-body (**t**,
get (car (last (*pos*)), cur-expr (butlast (*pos*), *body*)),
temp-alist,
table)
= lr-compile-body (**t**, cur-expr (*pos*, *body*), *temp-alist*, *table*))

THEOREM: good-posp1-cons-lessp-4-if-s-proper-exprp
((car (cur-expr (*pos*, *body*)) = `'if`)
∧ good-posp1 (*pos*, *body*)
∧ s-proper-exprp (**t**, *body*, *program-names*, *formals*, *temp-list*))
→ (good-posp1 (dv (*pos*, `1`), *body*)
∧ good-posp1 (dv (*pos*, `2`), *body*)
∧ good-posp1 (dv (*pos*, `3`), *body*))

THEOREM: lr-proper-exprp-flag-list-cons
lr-proper-exprp (`'list`, cons (*car*, *cdr*), *program-names*, *formals*, *temps*, *table*)
= (lr-proper-exprp (`'list`, *cdr*, *program-names*, *formals*, *temps*, *table*)
∧ lr-proper-exprp (**t**, *car*, *program-names*, *formals*, *temps*, *table*))

THEOREM: lr-proper-exprp-flag-list-nil
lr-proper-exprp (`'list`, **nil**, *program-names*, *formals*, *temps*, *table*)

THEOREM: length-3-cdr-cddr-not-nil
((length (*x*) = `3`) ∧ plistp (*x*))
→ ((cdddr (*x*) = **nil**) ∧ (cddr (*x*) ≠ **nil**) ∧ (cdr (*x*) ≠ **nil**))

THEOREM: lr-proper-exprp-flag-not-list-cons-if-helper
(*flag* ≠ `'list`)
→ (lr-proper-exprp (*flag*,
list (`'if`, *test*, *then*, *else*),
program-names,
formals,
temp-alist,
table)
= ((`'if` ∉ *program-names*)
∧ lr-proper-exprp (**t**,

test,
program-names,
formals,
temp-alist,
table)
∧ lr-proper-exprp (**t**,
then,
program-names,
formals,
temp-alist,
table)
∧ lr-proper-exprp (**t**,
else,
program-names,
formals,
temp-alist,
table)))

THEOREM: lr-proper-exprp-flag-not-list-cons-if
((*flag* ≠ `'list`)
∧ listp (cur-expr (*pos*, *body*))
∧ (car (cur-expr (*pos*, *body*)) = `'if`)
∧ good-posp1 (*pos*, *body*)
∧ lr-proper-exprp (**t**,
lr-compile-body (**t**,
cur-expr (dv (*pos*, `3`), *body*),
temp-alist,
table),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)
∧ lr-proper-exprp (**t**,
lr-compile-body (**t**,
cur-expr (dv (*pos*, `2`), *body*),
temp-alist,
table),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)
∧ lr-proper-exprp (**t**,
lr-compile-body (**t**,
cur-expr (dv (*pos*, `1`), *body*),

temp-alist,
table),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)
∧ s-proper-exprp (**t**, *body*, *program-names*, *formals*, strip-cars (*temp-alist*)))
→ lr-proper-exprp (*flag*,
cons (`'if`,
lr-compile-body (`'list`,
cdr (cur-expr (*pos*, *body*)),
temp-alist,
table)),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)

THEOREM: lr-proper-exprp-flag-not-list-cons-temp-fetch
((*flag* ≠ `'list`)
∧ good-posp1 (*pos*, *body*)
∧ listp (cur-expr (*pos*, *body*))
∧ (car (cur-expr (*pos*, *body*)) = S-TEMP-FETCH)
∧ s-proper-exprp (**t**, *body*, *program-names*, *formals*, strip-cars (*temp-alist*)))
→ lr-proper-exprp (*flag*,
list (identity (S-TEMP-FETCH),
x,
cdr (assoc (cadr (cur-expr (*pos*, *body*)), *temp-alist*))),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)

THEOREM: good-posp1-dv-1-temp-eval-test
(good-posp1 (*pos*, *body*)
∧ ((car (cur-expr (*pos*, *body*)) = S-TEMP-EVAL)
∨ (car (cur-expr (*pos*, *body*)) = S-TEMP-TEST))
∧ listp (cur-expr (*pos*, *body*)))
→ good-posp1 (dv (*pos*, `1`), *body*)

THEOREM: length-last-fact
(length (*x*) = `1`) → (last (*x*) = *x*)

THEOREM: lr-proper-exprp-flag-not-list-cons-temp-eval
((*flag* ≠ `'list`)

$\wedge$ listp (cur-expr (*pos*, *body*))
$\wedge$ good-posp1 (*pos*, *body*)
$\wedge$ (car (cur-expr (*pos*, *body*)) = S-TEMP-EVAL)
$\wedge$ lr-proper-exprp (**t**,
lr-compile-body (**t**,
cur-expr (dv (*pos*, 1), *body*),
temp-alist,
table),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)
$\wedge$ s-proper-exprp (**t**, *body*, *program-names*, *formals*, strip-cars (*temp-alist*)))
$\rightarrow$ lr-proper-exprp (*flag*,
list (identity (S-TEMP-EVAL),
lr-compile-body (**t**,
cadr (cur-expr (*pos*, *body*)),
temp-alist,
table),
cdr (assoc (cadr (cur-expr (*pos*, *body*)), *temp-alist*))),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)

THEOREM: lr-proper-exprp-flag-not-list-cons-temp-test
((*flag* $\neq$ 'list)
$\wedge$ listp (cur-expr (*pos*, *body*))
$\wedge$ good-posp1 (*pos*, *body*)
$\wedge$ (car (cur-expr (*pos*, *body*)) = S-TEMP-TEST)
$\wedge$ lr-proper-exprp (**t**,
lr-compile-body (**t**,
cur-expr (dv (*pos*, 1), *body*),
temp-alist,
table),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)
$\wedge$ s-proper-exprp (**t**, *body*, *program-names*, *formals*, strip-cars (*temp-alist*)))
$\rightarrow$ lr-proper-exprp (*flag*,
list (identity (S-TEMP-TEST),
lr-compile-body (**t**,
cadr (cur-expr (*pos*, *body*)),

temp-alist,
table),
cdr (assoc (cadr (cur-expr (*pos*, *body*)), *temp-alist*))),
program-names,
formals,
strip-cdrs (*temp-alist*),
table)

DEFINITION:
lr-data-seg-table-body-n (*n*, *body*, *data-seg*, *table*)
= **if** *n* ≃ `0` **then** cons (*data-seg*, *table*)
else lr-data-seg-table-body-n (*n* − 1,
cdr (*body*),
car (lr-data-seg-table-body (**t**,
car (*body*),
data-seg,
table)),
cdr (lr-data-seg-table-body (**t**,
car (*body*),
data-seg,
table))) **endif**

DEFINITION:
induct-hint-20 (*pos*, *body*, *data-seg*, *table*)
= **if** *pos* ≃ **nil then t**
elseif *body* ≃ **nil then f**
elseif car (*body*) = S-TEMP-FETCH **then f**
elseif (car (*body*) = S-TEMP-EVAL) ∨ (car (*body*) = S-TEMP-TEST)
then induct-hint-20 (cdr (*pos*), cadr (*body*), *data-seg*, *table*)
elseif car (*body*) = `'quote` **then f**
else induct-hint-20 (cdr (*pos*),
get (car (*pos*), *body*),
car (lr-data-seg-table-body-n (car (*pos*) − 1,
cdr (*body*),
data-seg,
table)),
cdr (lr-data-seg-table-body-n (car (*pos*) − 1,
cdr (*body*),
data-seg,
table))) **endif**

THEOREM: lr-data-seg-body-list-n
lr-data-seg-table-body (`'list`, *body*, *data-seg*, *table*)
= lr-data-seg-table-body-n (length (*body*), *body*, *data-seg*, *table*)

THEOREM: definedp-table-definedp-cdr-lr-data-seg-table-body-n
definedp (*object*, *table*)
→ definedp (*object*, cdr (lr-data-seg-table-body-n (*n*, *body*, *data-seg*, *table*)))

THEOREM: definedp-lr-data-seg-body-list-n-not-lessp
(definedp (*x*, cdr (lr-data-seg-table-body-n (*n*, *body*, *data-seg*, *table*)))
∧ (*m* ≮ *n*))
→ definedp (*x*, cdr (lr-data-seg-table-body-n (*m*, *body*, *data-seg*, *table*)))

THEOREM: lr-data-seg-table-body-add1-opener
(*n* < length (*body*))
→ (lr-data-seg-table-body-n (1 + *n*, *body*, *data-seg*, *table*)
= lr-data-seg-table-body (**t**,
get (*n*, *body*),
car (lr-data-seg-table-body-n (*n*,
body,
data-seg,
table)),
cdr (lr-data-seg-table-body-n (*n*,
body,
data-seg,
table))))

THEOREM: lr-data-seg-table-body-flag-t-flag-t
(definedp (*object*,
cdr (lr-data-seg-table-body (**t**,
get (*n*, *body*),
car (lr-data-seg-table-body-n (*n*,
body,
data-seg,
table)),
cdr (lr-data-seg-table-body-n (*n*,
body,
data-seg,
table)))))
∧ (*n* ∈ **N**)
∧ (*n* < length (*body*)))
→ definedp (*object*,
cdr (lr-data-seg-table-body (`'list`, *body*, *data-seg*, *table*)))

THEOREM: definedp-cadr-cur-expr-quote-lr-data-seg-table-body
(listp (cur-expr (*pos*, *body*))
∧ (car (cur-expr (*pos*, *body*)) = `'quote`)
∧ good-posp1 (*pos*, *body*))
→ definedp (cadr (cur-expr (*pos*, *body*)),
cdr (lr-data-seg-table-body (**t**, *body*, *data-seg*, *table*)))

THEOREM: definedp-cadr-cur-expr-quote-lr-data-seg-table-list
(listp (cur-expr (*pos*, s-body (*prog*)))
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) = 'quote)
 ∧ (*prog* ∈ *progs*)
 ∧ good-posp1 (*pos*, s-body (*prog*)))
→ definedp (cadr (cur-expr (*pos*, s-body (*prog*))),
 cdr (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))

THEOREM: definedp-cadr-cur-expr-quote-lr-data-seg-table
(listp (cur-expr (*pos*, s-body (*prog*)))
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) = 'quote)
 ∧ (*prog* ∈ *progs*)
 ∧ good-posp1 (*pos*, s-body (*prog*)))
→ definedp (cadr (cur-expr (*pos*, s-body (*prog*))),
 cdr (lr-data-seg-table (*progs*, *params*, *heap-size*)))

THEOREM: lr-proper-exprp-p-lr-compile-programs-helper-2
(listp (cur-expr (*pos*, s-body (*prog*)))
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) = 'quote)
 ∧ (*prog* ∈ *progs*)
 ∧ good-posp1 (*pos*, s-body (*prog*))
 ∧ (*heap-size* ≮ s-total-heap-reqs (*progs*, *params*, *heap-size*))
 ∧ s-restrictedp (*progs*, *params*))
→ (type (cdr (assoc (cadr (cur-expr (*pos*, s-body (*prog*))),
 cdr (lr-data-seg-table (*progs*, *params*, *heap-size*)))))
 = 'addr)

THEOREM: good-posp-dv-1-funcall-opened
(listp (cur-expr (*pos*, s-body (*prog*)))
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ 'if)
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ S-TEMP-EVAL)
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ S-TEMP-TEST)
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ S-TEMP-FETCH)
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ 'quote)
 ∧ good-posp1 (*pos*, s-body (*prog*)))
→ good-posp ('list, dv (*pos*, 1), s-body (*prog*))

THEOREM: s-restrict-subrps-list-lr-proper-get-t
(s-restrict-subrps ('list, cdr (*expr*))
 ∧ listp (*expr*)
 ∧ (*n* ≯ 0)
 ∧ (*n* < length (*expr*)))
→ s-restrict-subrps (t, get (*n*, *expr*))

THEOREM: s-restrict-subrps-t-lr-proper-get-t

((car (*body*) ≠ S-TEMP-FETCH)
 ∧ (car (*body*) ≠ S-TEMP-EVAL)
 ∧ (car (*body*) ≠ S-TEMP-TEST)
 ∧ (car (*body*) ≠ 'quote)
 ∧ listp (*body*)
 ∧ s-proper-exprp (**t**, *body*, *program-names*, *formals*, *temp-list*)
 ∧ (*n* ≉ 0)
 ∧ (*n* < length (*body*))
 ∧ s-restrict-subrps (**t**, *body*))
→ s-restrict-subrps (**t**, get (*n*, *body*))

EVENT: Disable s-restrict-subrps-list-lr-proper-get-t.

THEOREM: s-restrict-subrps-s-restrict-subrps-cur-expr
(s-restrict-subrps (**t**, *body*)
 ∧ s-proper-exprp (**t**, *body*, *program-names*, *formals*, *temp-list*)
 ∧ good-posp1 (*pos*, *body*))
→ s-restrict-subrps (**t**, cur-expr (*pos*, *body*))

THEOREM: lr-proper-exprp-flag-not-list-cons-funcall
((*flag* ≠ 'list)
 ∧ listp (cur-expr (*pos*, s-body (*prog*)))
 ∧ good-posp1 (*pos*, s-body (*prog*))
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ 'if)
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ S-TEMP-FETCH)
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ S-TEMP-EVAL)
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ S-TEMP-TEST)
 ∧ (car (cur-expr (*pos*, s-body (*prog*))) ≠ 'quote)
 ∧ lr-proper-exprp ('list,
 lr-compile-body ('list,
 cdr (cur-expr (*pos*, s-body (*prog*))),
 temp-alist,
 table),
 program-names,
 formals,
 temps,
 table)
 ∧ s-restrict-subrps (**t**, s-body (*prog*))
 ∧ s-proper-exprp (**t**, s-body (*prog*), *program-names*, *formals*, *temp-list*))
→ lr-proper-exprp (*flag*,
 cons (car (cur-expr (*pos*, s-body (*prog*))),
 lr-compile-body ('list,
 cdr (cur-expr (*pos*, s-body (*prog*))),
 temp-alist,

table)),
program-names,
formals,
temps,
table)

THEOREM: lr-proper-exprp-flag-not-list-not-listp
((*flag* ≠ 'list)
∧ (¬ listp (cur-expr (*pos*, *body*)))
∧ good-posp1 (*pos*, *body*)
∧ s-proper-exprp (**t**, *body*, *program-names*, *formals*, *temp-list*))
→ lr-proper-exprp (*flag*,
cur-expr (*pos*, *body*),
program-names,
formals,
temps,
table)

THEOREM: lr-proper-exprp-p-lr-compile-programs
(s-restrict-subrps (**t**, s-body (*prog*))
∧ (*prog* ∈ *progs*)
∧ subsetp (*progs*, *all-progs*)
∧ s-proper-exprp (**t**,
s-body (*prog*),
program-names,
formals,
strip-cars (*temp-alist*))
∧ good-posp (*flag*, *pos*, s-body (*prog*))
∧ (*heap-size* ≮ s-total-heap-reqs (*all-progs*, *params*, *heap-size*))
∧ s-restrictedp (*all-progs*, *params*))
→ lr-proper-exprp (*flag*,
lr-compile-body (*flag*,
if *flag* = 'list
then restn (car (last (*pos*)),
cur-expr (butlast (*pos*),
s-body (*prog*)))
else cur-expr (*pos*, s-body (*prog*)) **endif**,
temp-alist,
cdr (lr-data-seg-table (*all-progs*,
params,
heap-size))),
program-names,
formals,
strip-cdrs (*temp-alist*),
cdr (lr-data-seg-table (*all-progs*, *params*, *heap-size*)))

EVENT: Disable lr-proper-exprp-p-lr-compile-programs-helper-1.

THEOREM: all-litatoms-s-formals-member-s-programs-properp
((*prog* ∈ *progs*) ∧ s-programs-properp (*progs*, *program-names*))
→ all-litatoms (s-formals (*prog*))

THEOREM: s-restrict-subrps-s-body-member-s-restrict-subrps-progs
(s-restrict-subrps-progs (*progs*) ∧ (*prog* ∈ *progs*))
→ s-restrict-subrps (**t**, s-body (*prog*))

THEOREM: lr-proper-exprp-p-lr-compile-programs-flag-t
(s-restrict-subrps-progs (*all-progs*)
∧ (*prog* ∈ *all-progs*)
∧ s-programs-properp (*all-progs*, *program-names*)
∧ (*heap-size* ≮ s-total-heap-reqs (*all-progs*, *params*, *heap-size*))
∧ s-restrictedp (*all-progs*, *params*)
∧ (*temp-alist* = lr-make-temp-name-alist (s-temp-list (*prog*),
s-formals (*prog*))))
→ lr-proper-exprp (**t**,
lr-compile-body (**t**,
s-body (*prog*),
temp-alist,
cdr (lr-data-seg-table (*all-progs*,
params,
heap-size))),
program-names,
s-formals (*prog*),
strip-cdrs (*temp-alist*),
cdr (lr-data-seg-table (*all-progs*, *params*, *heap-size*)))

THEOREM: lr-programs-properp-1-lr-compile-programs
(s-programs-properp (*all-progs*, *program-names*)
∧ s-programs-okp (*progs*)
∧ s-restrict-subrps-progs (*all-progs*)
∧ subsetp (*progs*, *all-progs*)
∧ (*heap-size* ≮ s-total-heap-reqs (*all-progs*, *params*, *heap-size*))
∧ s-restrictedp (*all-progs*, *params*))
→ lr-programs-properp-1 (lr-compile-programs (*progs*,
cdr (lr-data-seg-table (*all-progs*,
params,
heap-size))),
program-names,
cdr (lr-data-seg-table (*all-progs*,
params,
heap-size)))

EVENT: Disable all-litatoms-s-formals-member-s-programs-properp.

THEOREM: subsetp-cdr
subsetp (cdr (*x*), *x*)

THEOREM: lr-programs-properp-s->lr-opened
(s-good-statep (*s*, *c*)
∧ s-restrict-subrps-progs (s-progs (*s*))
∧ (*heap-size* ≮ s-total-heap-reqs (s-progs (*s*), s-params (*s*), *heap-size*))
∧ s-restrictedp (s-progs (*s*), s-params (*s*))
∧ litatom (name (car (s-progs (*s*)))))
→ lr-programs-properp (s->lr1 (*s*,
p-state (*pc*,
ctrl-stk,
temp-stk,
anything,
data-seg,
max-ctrl,
max-temp,
word-size,
nil),
cdr (lr-data-seg-table (s-progs (*s*),
s-params (*s*),
heap-size))),
cdr (lr-data-seg-table (s-progs (*s*),
s-params (*s*),
heap-size)))

THEOREM: fall-off-proofp-append-cons-ret
fall-off-proofp (append (*instrs*, list (list (`'dl`, *label*, *comment*, `'(ret)`))))

THEOREM: proper-labeled-p-instructionsp-label-ret
litatom (*label*)
→ proper-labeled-p-instructionsp (list (list (`'dl`, *label*, *comment*, `'(ret)`)),
name,
p)

THEOREM: member-definedp-car
(*x* ∈ *y*) → definedp (car (*x*), *y*)

THEOREM: all-litatoms-s-formals-member-lr-programs-properp
((*prog* ∈ *prog-seg*) ∧ (¬ all-litatoms (formal-vars (*prog*))))
→ (¬ lr-programs-properp-1 (*prog-seg*, *program-names*, *table*))

THEOREM: properp-p-temp-var-dclps-member-lr-programs-properp
((*prog* ∈ *progs*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ lr-programs-properp-1 (*progs*, strip-logic-fnames (cdr (*prog-seg*)), *table*))
→ proper-p-temp-var-dclsp (temp-var-dcls (*prog*),
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

THEOREM: member-f-definedp-0
(**f** ∈ *alist*) → definedp (0, *alist*)

THEOREM: member-no-duplicatesp-assoc-equal
((*x* ∈ *alist*) ∧ no-duplicatesp (strip-cars (*alist*)) ∧ good-alistp (*alist*))
→ (assoc (car (*x*), *alist*) = *x*)

EVENT: Disable member-f-definedp-0.

THEOREM: lr-proper-exprp-program-body-not-listp
(¬ listp (*prog*))
→ (¬ lr-proper-exprp (**t**, program-body (*prog*), *pnames*, *formals*, *temps*, *table*))

THEOREM: good-alistp-lr-programs-properp
(¬ good-alistp (*prog-seg*))
→ (¬ lr-programs-properp-1 (*prog-seg*, *program-names*, *table*))

EVENT: Disable lr-proper-exprp-program-body-not-listp.

THEOREM: proper-p-prog-segmentp-comp-programs-1-helper
(lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ lr-programs-properp-1 (*prog-seg*,
strip-logic-fnames (cdr (*prog-seg*)),
table)
 ∧ all-user-fnamesp (cdr (strip-cars (*prog-seg*)))
 ∧ lr-s-similar-const-table (*table*, *data-seg*)
 ∧ (*prog* ∈ *prog-seg*)

∧ no-duplicatesp (strip-cars (*prog-seg*)))
→ proper-labeled-p-instructionsp (comp-body-1 (**t**, program-body (*prog*), 0),
car (*prog*),
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl,
max-temp,
word-size,
psw))

THEOREM: proper-p-prog-segmentp-comp-programs-1
(lr-proper-heapp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ lr-programs-properp-1 (*prog-seg*,
strip-logic-fnames (cdr (*prog-seg*)),
table)
∧ all-user-fnamesp (cdr (strip-cars (*prog-seg*)))
∧ lr-s-similar-const-table (*table*, *data-seg*)
∧ no-duplicatesp (strip-cars (*prog-seg*))
∧ subsetp (*progs*, *prog-seg*)
∧ all-litatoms (strip-cars (*progs*)))
→ proper-p-prog-segmentp (comp-programs-1 (*progs*),
p-state (*pc*,
ctrl-stk,
temp-stk,
comp-programs (*prog-seg*),
data-seg,
max-ctrl-stk-size,
max-temp-stk-size,
word-size,
psw))

EVENT: Disable all-litatoms-s-formals-member-lr-programs-properp.

THEOREM: proper-p-instructionp-set-global
adpp (cons (*name*, 0), p-data-segment (*l*))
→ proper-p-instructionp (list ('set-global, *name*), *x*, lr->p (*l*))

THEOREM: proper-p-programp-append-car-prog-segment
(lr-programs-properp-1 (*prog-seg*, strip-logic-fnames (cdr (*prog-seg*)), *table*)
∧ definedp (area-name (p-pc (*l*)), p-prog-segment (*l*))

$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ lr-proper-p-areasp (p-data-segment (l))
$\wedge$ lr-s-similar-const-table ($table$, p-data-segment (l))
$\wedge$ adpp (untag (LR-ANSWER-ADDR), p-data-segment (l))
$\wedge$ ($prog\text{-}seg$ = p-prog-segment (l))
$\wedge$ all-litatoms (strip-cars ($prog\text{-}seg$))
$\wedge$ all-user-fnamesp (strip-cars (cdr (p-prog-segment (l))))
$\rightarrow$ proper-p-programp (cons (name (car ($prog\text{-}seg$)),
cons (formal-vars (car ($prog\text{-}seg$)),
cons (temp-var-dcls (car ($prog\text{-}seg$)),
append (label-instrs (comp-body-1 (**t**,
program-body (car ($prog\text{-}seg$)),
0),
0),
list (list (`'dl`,
lr-make-label ($n1$),
$coment1$,
list (`'set-global`,
identity (area-name (LR-ANSWER-ADDR)))),
list (`'dl`,
lr-make-label ($n2$),
$comment2$,
`'(ret)`))))))),
lr->p (l))

THEOREM: all-litatoms-all-user-fnamesp-plistp
(all-user-fnamesp ($list$) $\wedge$ plistp ($list$)) $\rightarrow$ all-litatoms ($list$)

THEOREM: plistp-strip-cars
plistp (strip-cars (x))

THEOREM: lr-programs-properp-all-user-fnamesp-strip-cars-cdr
lr-programs-properp (l, $table$)
$\rightarrow$ all-user-fnamesp (strip-cars (cdr (p-prog-segment (l))))

THEOREM: lr-programs-properp-caar-main
lr-programs-properp (l, $table$) $\rightarrow$ (caar (p-prog-segment (l)) = `'main`)

THEOREM: proper-p-prog-segmentp-p-runtime-support-programs
(lr-programs-properp (l, $table$)
$\wedge$ (p-word-size (l) $\in$ **N**)
$\wedge$ (p-word-size (l) $\not<$ S-MAX-SUBR-REQS)
$\wedge$ lr-proper-heapp (p-data-segment (l))
$\wedge$ lr-proper-p-areasp (p-data-segment (l)))
$\rightarrow$ proper-p-prog-segmentp (P-RUNTIME-SUPPORT-PROGRAMS, lr->p (l))

THEOREM: proper-p-prog-segmentp-comp-programs
(lr-proper-heapp (p-data-segment (*l*))
∧ lr-proper-p-areasp (p-data-segment (*l*))
∧ (*progs* = p-prog-segment (*l*))
∧ lr-programs-properp (*l*, *table*)
∧ lr-s-similar-const-table (*table*, p-data-segment (*l*))
∧ no-duplicatesp (strip-cars (p-prog-segment (*l*)))
∧ (p-word-size (*l*) ∈ **N**)
∧ (p-word-size (*l*) ≮ S-MAX-SUBR-REQS)
∧ adpp (untag (LR-ANSWER-ADDR), p-data-segment (*l*)))
→ proper-p-prog-segmentp (comp-programs (*progs*), lr->p (*l*))

EVENT: Disable lr-programs-properp-caar-main.

THEOREM: no-duplicatesp-remove-duplicates
no-duplicatesp (remove-duplicates (*x*))

THEOREM: all-p-objectps-put
(all-p-objectps (*lst*, *p*) ∧ p-objectp (*value*, *p*) ∧ (*offset* < length (*lst*)))
→ all-p-objectps (put (*value*, *offset*, *lst*), *p*)

THEOREM: proper-p-data-segmentp-deposit-helper
(proper-p-data-segmentp (*data-seg*, *p*)
∧ (offset (*addr*) < length (cdr (assoc (area-name (*addr*), *data-seg*))))
∧ p-objectp (*value*, *p*))
→ proper-p-data-segmentp (deposit (*value*, *addr*, *data-seg*), *p*)

THEOREM: proper-p-data-segmentp-deposit
(proper-p-data-segmentp (*data-seg*, *p*)
∧ adpp (untag (*addr*), *data-seg*)
∧ p-objectp (*value*, *p*))
→ proper-p-data-segmentp (deposit (*value*, *addr*, *data-seg*), *p*)

THEOREM: all-p-objectps-get
(all-p-objectps (*lst*, *p*)
∧ (*offset* < length (*lst*))
∧ all-p-objectps (*lst*, *p*))
→ p-objectp (get (*offset*, *lst*), *p*)

THEOREM: proper-p-data-segmentp-fetch
(proper-p-data-segmentp (*data-seg*, *p*)
∧ (offset (*addr*) < length (cdr (assoc (area-name (*addr*), *data-seg*))))
∧ definedp (area-name (*addr*), *data-seg*))
→ p-objectp (fetch (*addr*, *data-seg*), *p*)

THEOREM: lr-s-similar-const-table-p-objectp-definedp
(lr-s-similar-const-table (*table*, *data-seg*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ definedp (**f**, *table*)
 ∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
 lr-free-list-nodes (lr-max-node (*data-seg*),
 data-seg),
 data-seg)
 ≮ s-heap-reqs (*flag*, *object2*, *data-seg*, *table*))
 ∧ definedp (*object1*, cdr (lr-compile-quote (*flag*, *object2*, *data-seg*, *table*)))
 ∧ same-signature (p-data-segment (*p*), *data-seg*)
 ∧ s-restricted-objectp (*flag*, *object2*))
 → p-objectp (cdr (assoc (*object1*,
 cdr (lr-compile-quote (*flag*, *object2*, *data-seg*, *table*)))),
 p)

THEOREM: lessp-x-sub1-facts
$(x < ((y - 1) - 1)) \rightarrow ((x < y) \wedge (x < (y - 1)))$

THEOREM: proper-p-data-segmentp-deposit-a-list-cons
((area-name (*addr*) = LR-HEAP-NAME)
 ∧ adpp (untag (*addr*), *data-seg*)
 ∧ ((offset (*addr*) + 3) < length (cdr (assoc (LR-HEAP-NAME, *data-seg*))))
 ∧ p-objectp (*a*, *p*)
 ∧ p-objectp (*b*, *p*)
 ∧ p-objectp (*c*, *p*)
 ∧ p-objectp (*d*, *p*)
 ∧ proper-p-data-segmentp (*data-seg*, *p*))
 → proper-p-data-segmentp (deposit-a-list (list (*a*, *b*, *c*, *d*), *addr*, *data-seg*), *p*)

THEOREM: proper-p-data-segmentp-deposit-a-list-cons-cons
let *ds-tab* **be** lr-compile-quote (`'list`, list (*x*, *y*), *data-seg*, *table*),
 count-nodes **be** lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
 lr-free-list-nodes (lr-max-node (*data-seg*),
 data-seg),
 data-seg)
in
(lr-proper-p-areasp (*data-seg*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ lr-s-similar-const-table (*table*, *data-seg*)
 ∧ definedp (**f**, *table*)
 ∧ ((*count-nodes* − 1)
 ≮ s-heap-reqs (`'list`, list (*x*, *y*), *data-seg*, *table*))
 ∧ (*count-nodes* ≠ `0`)

$\wedge$ proper-p-data-segmentp (car (*ds-tab*), *p*)
$\wedge$ ((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) − 1)
$\not<$ (offset (fetch (LR-FP-ADDR, car (*ds-tab*)))
+ LR-NODE-SIZE))
$\wedge$ (p-word-size (*p*) $\in$ **N**)
$\wedge$ (p-word-size (*p*) $\not<$ S-MAX-SUBR-REQS)
$\wedge$ same-signature (p-data-segment (*p*), *data-seg*)
$\wedge$ s-restricted-objectp (`'list`, list (*x*, *y*)))
$\rightarrow$ proper-p-data-segmentp (deposit-a-list (list (identity (tag (`'nat`, LR-CONS-TAG)),
identity (tag (`'nat`, `1`)),
cdr (assoc (*x*, cdr (*ds-tab*))),
cdr (assoc (*y*, cdr (*ds-tab*)))),
fetch (identity (LR-FP-ADDR), car (*ds-tab*)),
car (*ds-tab*)),
p) **endlet**

THEOREM: proper-p-data-segmentp-deposit-a-list-cons-numberp
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ (p-word-size (*p*) $\not<$ log (`2`, *number*))
$\wedge$ (*number* $\in$ **N**)
$\wedge$ proper-p-data-segmentp (*data-seg*, *p*)
$\wedge$ ((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) − 1)
$\not<$ (offset (fetch (LR-FP-ADDR, *data-seg*)) + LR-NODE-SIZE))
$\wedge$ (p-word-size (*p*) $\in$ **N**)
$\wedge$ (p-word-size (*p*) $\not<$ S-MAX-SUBR-REQS)
$\wedge$ same-signature (p-data-segment (*p*), *data-seg*))
$\rightarrow$ proper-p-data-segmentp (deposit-a-list (list (identity (tag (`'nat`, LR-ADD1-TAG)),
identity (tag (`'nat`, `1`)),
tag (`'nat`, *number*),
identity (LR-UNDEF-ADDR)),
fetch (identity (LR-FP-ADDR), *data-seg*),
data-seg),
p)

THEOREM: proper-p-data-segmentp-deposit-a-list-cons-truep
(lr-proper-p-areasp (*data-seg*)

$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ proper-p-data-segmentp (*data-seg*, *p*)
$\wedge$ ((length (cdr (assoc (LR-HEAP-NAME, *data-seg*))) − 1)
$\not<$ (offset (fetch (LR-FP-ADDR, *data-seg*)) + LR-NODE-SIZE))
$\wedge$ (p-word-size (*p*) $\in$ **N**)
$\wedge$ (p-word-size (*p*) $\not<$ log (2, LR-TRUE-TAG))
$\wedge$ same-signature (p-data-segment (*p*), *data-seg*))
$\rightarrow$ proper-p-data-segmentp (deposit-a-list (list (identity (tag ('nat, LR-TRUE-TAG)),
identity (tag ('nat, 1)),
identity (LR-UNDEF-ADDR),
identity (LR-UNDEF-ADDR)),
fetch (identity (LR-FP-ADDR),
data-seg),
data-seg),
p)

EVENT: Disable proper-p-data-segmentp-deposit-a-list-cons.

THEOREM: s-ws-reqs-flag-not-list-t
((*flag* $\neq$ 'list) $\wedge$ ($\neg$ definedp (**t**, *table*)))
$\rightarrow$ (s-ws-reqs (*flag*, **t**, *data-seg*, *table*) = identity (log (2, LR-TRUE-TAG)))

THEOREM: lr-compile-quote-preserves-proper-p-data-segmentp
(lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ s-restricted-objectp (*flag*, *object*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (p-word-size (*p*) $\in$ **N**)
$\wedge$ (p-word-size (*p*) $\not<$ s-ws-reqs (*flag*, *object*, *data-seg*, *table*))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*))
$\wedge$ proper-p-data-segmentp (*data-seg*, *p*)
$\wedge$ same-signature (p-data-segment (*p*), *data-seg*))
$\rightarrow$ proper-p-data-segmentp (car (lr-compile-quote (*flag*,
object,
data-seg,
table)),
p)

THEOREM: lr-data-seg-table-body-preserves-proper-p-data-segmentp
(lr-s-similar-const-table (*table*, *data-seg*)
 ∧ s-data-seg-body-restrictedp (*flag*, *body*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ proper-p-data-segmentp (*data-seg*, *p*)
 ∧ definedp (**f**, *table*)
 ∧ (p-word-size (*p*) ∈ **N**)
 ∧ (p-word-size (*p*) ⊀ s-ws-reqs-body (*flag*, *body*, *data-seg*, *table*))
 ∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
 lr-free-list-nodes (lr-max-node (*data-seg*),
 data-seg),
 data-seg)
 ⊀ s-heap-reqs-body (*flag*, *body*, *data-seg*, *table*))
 ∧ same-signature (p-data-segment (*p*), *data-seg*))
→ proper-p-data-segmentp (car (lr-data-seg-table-body (*flag*,
 body,
 data-seg,
 table)),
 p)

THEOREM: lr-data-seg-table-list-preserves-proper-p-data-segmentp
(lr-s-similar-const-table (*table*, *data-seg*)
 ∧ s-data-seg-list-restrictedp (*progs*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ proper-p-data-segmentp (*data-seg*, *p*)
 ∧ definedp (**f**, *table*)
 ∧ (p-word-size (*p*) ∈ **N**)
 ∧ (p-word-size (*p*) ⊀ s-ws-reqs-list (*progs*, *data-seg*, *table*))
 ∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
 lr-free-list-nodes (lr-max-node (*data-seg*),
 data-seg),
 data-seg)
 ⊀ s-heap-reqs-list (*progs*, *data-seg*, *table*))
 ∧ same-signature (p-data-segment (*p*), *data-seg*))
→ proper-p-data-segmentp (car (lr-data-seg-table-list (*progs*,
 data-seg,
 table)),
 p)

THEOREM: lr-init-data-seg-table-preserves-proper-p-data-segmentp
(lr-s-similar-const-table (*table*, *data-seg*)
 ∧ s-init-data-seg-restrictedp (*params*)

$$
\begin{array}{ll}
\wedge & \text{lr-proper-heapp}\,(\textit{data-seg}) \\
\wedge & \text{lr-proper-p-areasp}\,(\textit{data-seg}) \\
\wedge & \text{proper-p-data-segmentp}\,(\textit{data-seg},\ p) \\
\wedge & \text{definedp}\,(\mathbf{f},\ \textit{table}) \\
\wedge & (\text{p-word-size}\,(p) \in \mathbf{N}) \\
\wedge & (\text{p-word-size}\,(p) \not< \text{s-init-ws-reqs}\,(\textit{params},\ \textit{data-seg},\ \textit{table})) \\
\wedge & (\text{lr-count-free-nodes}\,(\text{fetch}\,(\textsc{lr-fp-addr},\ \textit{data-seg}), \\
 & \qquad\qquad \text{lr-free-list-nodes}\,(\text{lr-max-node}\,(\textit{data-seg}), \\
 & \qquad\qquad\qquad\qquad \textit{data-seg}), \\
 & \qquad\qquad \textit{data-seg}) \\
 & \quad \not< \quad \text{s-init-heap-reqs}\,(\textit{params},\ \textit{data-seg},\ \textit{table})) \\
\wedge & \text{same-signature}\,(\text{p-data-segment}\,(p),\ \textit{data-seg})) \\
\rightarrow & \text{proper-p-data-segmentp}\,(\text{car}\,(\text{lr-init-data-seg-table}\,(\textit{params}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{data-seg}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{table})), \\
 & \qquad\qquad\qquad\qquad p)
\end{array}
$$

THEOREM: all-p-objectps-lr-init-heap-contents-helper-helper

$$
\begin{array}{ll}
 & (\textit{heap-size} \not\simeq \texttt{0}) \\
\rightarrow & ((\text{identity}\,(\textsc{lr-node-size}) \\
 & \quad + \quad x \\
 & \quad + \quad (\text{identity}\,(\textsc{lr-node-size}) * (\textit{heap-size} - 1))) \\
 & \quad = \quad (x + (\text{identity}\,(\textsc{lr-node-size}) * \textit{heap-size})))
\end{array}
$$

THEOREM: all-p-objectps-lr-init-heap-contents-helper

$$
\begin{array}{ll}
 & ((\text{p-word-size}\,(p) \in \mathbf{N}) \\
\wedge & (\text{p-word-size}\,(p) \not< \textsc{s-max-subr-reqs}) \\
\wedge & (\textit{heap-size} \not< \texttt{4}) \\
\wedge & \text{lr-proper-p-areasp}\,(\text{p-data-segment}\,(p)) \\
\wedge & \text{lr-minimum-heapp}\,(\text{p-data-segment}\,(p)) \\
\wedge & (\text{type}\,(\textit{addr}) = \texttt{'addr}) \\
\wedge & (\text{cddr}\,(\textit{addr}) = \mathbf{nil}) \\
\wedge & \text{listp}\,(\textit{addr}) \\
\wedge & (\text{offset}\,(\textit{addr}) \in \mathbf{N}) \\
\wedge & \text{lr-boundary-nodep}\,(\textit{addr}) \\
\wedge & (\text{area-name}\,(\textit{addr}) = \textsc{lr-heap-name}) \\
\wedge & \text{definedp}\,(\textsc{lr-heap-name},\ \text{p-data-segment}\,(p)) \\
\wedge & (((\textsc{lr-node-size} * \textit{heap-size}) + \text{offset}\,(\textit{addr})) \\
 & \quad < \quad \text{length}\,(\text{cdr}\,(\text{assoc}\,(\textsc{lr-heap-name},\ \text{p-data-segment}\,(p))))) \\
\rightarrow & \text{all-p-objectps}\,(\text{lr-init-heap-contents}\,(\textit{addr},\ \textit{heap-size}),\ p)
\end{array}
$$

EVENT: Disable all-p-objectps-lr-init-heap-contents-helper-helper.

THEOREM: all-p-objectps-lr-init-heap-contents

((p-word-size (*p*) ∈ **N**)
∧ (p-word-size (*p*) ∉ S-MAX-SUBR-REQS)
∧ (*heap-size* ≮ 4)
∧ lr-proper-p-areasp (p-data-segment (*p*))
∧ same-signature (p-data-segment (*p*), lr-init-data-seg (*heap-size*)))
→ all-p-objectps (lr-init-heap-contents (identity (tag ('addr,
cons (LR-HEAP-NAME,
0))),
heap-size),
p)

THEOREM: proper-p-data-segmentp-lr-init-data-seg-helper
((p-word-size (*p*) ∈ **N**)
∧ (p-word-size (*p*) ∉ S-MAX-SUBR-REQS)
∧ (*heap-size* ≮ 4)
∧ lr-proper-p-areasp (p-data-segment (*p*))
∧ same-signature (p-data-segment (*p*), lr-init-data-seg (*heap-size*)))
→ proper-p-data-segmentp (deposit-a-list (list (tag ('nat, LR-FALSE-TAG),
tag ('nat, 1),
LR-UNDEF-ADDR,
LR-UNDEF-ADDR),
LR-F-ADDR,
deposit-a-list (list (tag ('nat,
LR-UNDEFINED-TAG),
tag ('nat,
1),
LR-UNDEF-ADDR,
LR-UNDEF-ADDR),
LR-UNDEF-ADDR,
list (list (area-name (LR-FP-ADDR),
add-addr (LR-F-ADDR,
LR-NODE-SIZE)),
list (area-name (LR-ANSWER-ADDR),
tag ('nat,
0)),
cons (LR-HEAP-NAME,
lr-init-heap-contents (tag ('addr,
cons (LR-HEAP
0)),
heap-size))))),
p)

THEOREM: proper-p-data-segmentp-lr-init-data-seg
((p-word-size (*p*) ∈ **N**)

$\wedge$ (p-word-size (p) $\not<$ S-MAX-SUBR-REQS)
$\wedge$ (*heap-size* $\not<$ 4)
$\wedge$ lr-proper-p-areasp (p-data-segment (p))
$\wedge$ same-signature (p-data-segment (p), lr-init-data-seg (*heap-size*)))
$\rightarrow$ proper-p-data-segmentp (lr-init-data-seg (*heap-size*), p)

THEOREM: proper-p-data-segmentp-lr-init-data-seg-compile-t0
((p-word-size (p) $\in$ **N**)
$\wedge$ (p-word-size (p) $\not<$ S-MAX-SUBR-REQS)
$\wedge$ (*heap-size* $\not<$ 4)
$\wedge$ lr-proper-p-areasp (p-data-segment (p))
$\wedge$ same-signature (p-data-segment (p), lr-init-data-seg (*heap-size*)))
$\rightarrow$ proper-p-data-segmentp (car (lr-compile-quote (`'list`,
list (**t**, `0`),
lr-init-data-seg (*heap-size*),
list (cons (**f**, *addr*)))),
p)

THEOREM: same-signature-car-lr-data-seg-table-list-reducer
(lr-proper-free-listp (*data-seg2*)
$\wedge$ lr-proper-p-areasp (*data-seg2*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg2*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg2*)))
$\rightarrow$ (same-signature (*data-seg1*,
car (lr-data-seg-table-list (*progs*, *data-seg2*, *table*)))
$\leftrightarrow$ same-signature (*data-seg1*, *data-seg2*))

THEOREM: same-signature-car-lr-init-data-seg-table-reducer
(lr-proper-free-listp (*data-seg2*)
$\wedge$ lr-proper-p-areasp (*data-seg2*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg2*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg2*)))
$\rightarrow$ (same-signature (*data-seg1*,
car (lr-init-data-seg-table (*params*, *data-seg2*, *table*)))
$\leftrightarrow$ same-signature (*data-seg1*, *data-seg2*))

THEOREM: same-signature-car-lr-compile-quote-reducer
(lr-proper-free-listp (*data-seg2*)
$\wedge$ lr-proper-p-areasp (*data-seg2*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg2*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg2*)))
$\rightarrow$ (same-signature (*data-seg1*,
car (lr-compile-quote (*flag*, *object*, *data-seg2*, *table*)))
$\leftrightarrow$ same-signature (*data-seg1*, *data-seg2*))

THEOREM: proper-p-data-segmentp-lr-data-seg-table
((p-word-size (*p*) ∈ **N**)
 ∧ (p-word-size (*p*) ≮ S-MAX-SUBR-REQS)
 ∧ lr-proper-p-areasp (p-data-segment (*p*))
 ∧ s-restrictedp (*progs*, *params*)
 ∧ (*heap-size* ≮ s-total-heap-reqs (*progs*, *params*, *heap-size*))
 ∧ (p-word-size (*p*) ≮ s-total-ws-reqs (*progs*, *params*, *heap-size*))
 ∧ same-signature (p-data-segment (*p*),
                   car (lr-data-seg-table (*progs*, *params*, *heap-size*))))
→ proper-p-data-segmentp (car (lr-data-seg-table (*progs*, *params*, *heap-size*)),
                           p)

THEOREM: adpp-untag-answer-addr-car-lr-data-seg-table
adpp (identity (untag (LR-ANSWER-ADDR)),
      car (lr-data-seg-table (*progs*, *params*, *heap-size*)))

THEOREM: program-body-assoc-cdr-lr-compile-programs
(*name* ≠ caar (*progs*))
→ (program-body (assoc (*name*, cdr (lr-compile-programs (*progs*, *table*))))
   = lr-compile-body (**t**,
                      s-body (assoc (*name*, cdr (*progs*))),
                      lr-make-temp-name-alist (s-temp-list (assoc (*name*,
                                                                  cdr (*progs*))),
                                               s-formals (assoc (*name*,
                                                                 cdr (*progs*)))),
                      table))

THEOREM: s-total-ws-reqs-not-lessp-s-max-subr-reqs
(*word-size* ≮ s-total-ws-reqs (*progs*, *params*, *heap-size*))
→ (*word-size* ≮ S-MAX-SUBR-REQS)

THEOREM: proper-p-statep-lr->p-s->lr
(s-good-statep (*s*, *c*)
 ∧ all-litatoms (strip-cars (s-params (*s*)))
 ∧ (*max-ctrl* ≮ (2 + length (s-params (*s*)) + length (s-temps (*s*))))
 ∧ (*word-size* ∈ **N**)
 ∧ (*max-temp* < exp (2, *word-size*))
 ∧ (*max-ctrl* < exp (2, *word-size*))
 ∧ (*max-temp* ∈ **N**)
 ∧ (*max-ctrl* ∈ **N**)
 ∧ (*heap-size* ≮ s-total-heap-reqs (s-progs (*s*), s-params (*s*), *heap-size*))
 ∧ s-restrict-subrps-progs (s-progs (*s*))
 ∧ litatom (name (car (s-progs (*s*))))
 ∧ no-duplicatesp (strip-cars (s-progs (*s*)))
 ∧ (strip-cars (s-params (*s*)) = s-formals (s-prog (*s*)))

```
  ∧   s-restrictedp (s-progs (s), s-params (s))
  ∧   (word-size ≮ s-total-ws-reqs (s-progs (s), s-params (s), heap-size)))
  →   proper-p-statep (lr->p (s->lr (s, heap-size, max-ctrl, max-temp, word-size)))

THEOREM: lr-programs-properp-s->lr
 (s-good-statep (s, c)
  ∧   s-restrict-subrps-progs (s-progs (s))
  ∧   s-restrictedp (s-progs (s), s-params (s))
  ∧   (heap-size ≮ s-total-heap-reqs (s-progs (s), s-params (s), heap-size))
  ∧   litatom (name (car (s-progs (s)))))
  →   lr-programs-properp (s->lr (s, heap-size, max-ctrl, max-temp, word-size),
                                 cdr (lr-data-seg-table (s-progs (s),
                                                         s-params (s),
                                                         heap-size)))

DEFINITION:
lr-total-heap-reqs (expr, alist, program-names, heap-size, c)
=   (s-total-heap-reqs (s-progs (logic->s (expr, alist, program-names)),
                        alist,
                        heap-size)
      +   s-eval-heap-r (t, logic->s (expr, alist, program-names), c))

DEFINITION:
lr-max-ctrl-reqs (expr, alist, program-names, c)
=   (2
      +   length (s-params (logic->s (expr, alist, program-names)))
      +   length (s-temps (logic->s (expr, alist, program-names)))
      +   s-eval-ctrl-r (t, logic->s (expr, alist, program-names), c))

DEFINITION:
lr-max-temp-reqs (expr, alist, program-names, c)
=   s-eval-temp-r (t, logic->s (expr, alist, program-names), c)

DEFINITION:
lr-max-word-size-reqs (expr, alist, program-names, heap-size, c)
=   max (s-total-ws-reqs (s-progs (logic->s (expr, alist, program-names)),
                          s-params (logic->s (expr, alist, program-names)),
                          heap-size),
         s-eval-ws-r (t, logic->s (expr, alist, program-names), c))

THEOREM: lr-eval-s-eval-equivalence-s->lr
 let s-lr be s->lr (s, heap-size, max-ctrl, max-temp, word-size)
 in
 (proper-p-statep (lr->p (s-lr))
  ∧   lr-programs-properp (s-lr,
```

cdr (lr-data-seg-table (s-progs (*s*),
s-params (*s*),
heap-size)))
∧ lr-s-similar-statesp (s-params (*s*),
s-temps (*s*),
s-lr,
cdr (lr-data-seg-table (s-progs (*s*),
s-params (*s*),
heap-size)))
∧ s-restrictedp (s-progs (*s*), s-params (*s*))
∧ (*heap-size* ≮ s-total-heap-reqs (s-progs (*s*),
s-params (*s*),
heap-size))
∧ s-good-statep (*s*, *c*)
∧ (p-psw (lr-eval (**t**, *s-lr*, *c*)) = 'run)
∧ (s-pname (*s*) = name (car (s-progs (*s*))))
∧ (s-pos (*s*) = **nil**))
→ lr-valp (s-ans (s-eval (**t**, *s*, *c*)),
car (p-temp-stk (lr-eval (**t**, *s-lr*, *c*))),
p-data-segment (lr-eval (**t**, *s-lr*, *c*))) **endlet**

THEOREM: same-signature-car-lr-data-seg-table
(*heap-size* ≮ s-total-heap-reqs (*progs*, *params*, *heap-size*))
→ same-signature (lr-init-data-seg (*heap-size*),
car (lr-data-seg-table (*progs*, *params*, *heap-size*)))

THEOREM: lr-max-node-car-lr-data-seg-table
(*heap-size* ≮ s-total-heap-reqs (*progs*, *params*, *heap-size*))
→ (lr-max-node (car (lr-data-seg-table (*progs*, *params*, *heap-size*)))
= tag ('addr,
cons (identity (LR-HEAP-NAME),
identity (LR-NODE-SIZE) ∗ *heap-size*)))

THEOREM: lr-count-free-nodes-lr-data-seg-table-list-s-heap-reqs-help
(lr-proper-free-listp (*data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ definedp (LR-HEAP-NAME, *data-seg*)
∧ lr-minimum-heapp (*data-seg*)
∧ lr-boundary-nodep (lr-max-node (*data-seg*))
∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
≮ s-heap-reqs-body (**t**, *body*, *data-seg*, *table*)))
→ ((s-heap-reqs-body (**t**, *body*, *data-seg*, *table*)

$+$ lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
car (lr-data-seg-table-body (**t**,
body,
data-seg,
table))),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (lr-data-seg-table-body (**t**,
body,
data-seg,
table))),
car (lr-data-seg-table-body (**t**,
body,
data-seg,
table))))
$=$ lr-count-free-nodes (fetch (identity (LR-FP-ADDR), *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg))

THEOREM: lr-count-free-nodes-lr-data-seg-table-list-s-heap-reqs
(lr-proper-free-listp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ definedp (LR-HEAP-NAME, *data-seg*)
$\wedge$ lr-minimum-heapp (*data-seg*)
$\wedge$ lr-boundary-nodep (lr-max-node (*data-seg*))
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-list (*progs*, *data-seg*, *table*)))
$\rightarrow$ ((lr-count-free-nodes (fetch (LR-FP-ADDR,
car (lr-data-seg-table-list (*progs*,
data-seg,
table))),
lr-free-list-nodes (lr-max-node (*data-seg*),
car (lr-data-seg-table-list (*progs*,
data-seg,
table))),
car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))
$+$ s-heap-reqs-list (*progs*, *data-seg*, *table*))
$=$ lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg))

THEOREM: lr-max-node-car-lr-data-seg-table-list
(lr-proper-free-listp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ definedp (LR-HEAP-NAME, *data-seg*)
 ∧ lr-boundary-nodep (lr-max-node (*data-seg*)))
→ (lr-max-node (car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))
 = lr-max-node (*data-seg*))

THEOREM: lr-count-free-nodes-lr-data-seg-table-list-s-heap-reqs-1
(lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
 lr-free-list-nodes (lr-max-node (*data-seg*),
 data-seg),
 data-seg)
 ⊀ s-heap-reqs-list (*progs*, *data-seg*, *table*))
 ∧ (*max-addr* = lr-max-node (car (lr-data-seg-table-list (*progs*,
 data-seg,
 table)))))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
 car (lr-data-seg-table-list (*progs*,
 data-seg,
 table))),
 lr-free-list-nodes (*max-addr*,
 car (lr-data-seg-table-list (*progs*,
 data-seg,
 table))),
 car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))
 = (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
 lr-free-list-nodes (lr-max-node (*data-seg*),
 data-seg),
 data-seg)
 − s-heap-reqs-list (*progs*, *data-seg*, *table*)))

THEOREM: lr-count-free-nodes-lr-init-data-seg-table-s-heap-reqs-1
(lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
 lr-free-list-nodes (lr-max-node (*data-seg*),
 data-seg),
 data-seg)
 ⊀ s-init-heap-reqs (*params*, *data-seg*, *table*))
 ∧ (*max-addr* = lr-max-node (car (lr-init-data-seg-table (*params*,
 data-seg,

table)))))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
car (lr-init-data-seg-table (*params*,
data-seg,
table))),
lr-free-list-nodes (*max-addr*,
car (lr-init-data-seg-table (*params*,
data-seg,
table))),
car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))
= (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
− s-init-heap-reqs (*params*, *data-seg*, *table*)))

THEOREM: not-lessp-lr-count-free-nodes-lr-data-seg-table-heap-r
((*heap-size* ≮ (s-total-heap-reqs (*progs*, *params*, *heap-size*) + *x*))
∧ s-restrictedp (*progs*, *params*))
→ (lr-count-free-nodes (fetch (identity (LR-FP-ADDR),
car (lr-data-seg-table (*progs*,
params,
heap-size))),
lr-free-list-nodes (lr-max-node (car (lr-data-seg-table (*progs*,
params,
heap-size))),
car (lr-data-seg-table (*progs*,
params,
heap-size))),
car (lr-data-seg-table (*progs*, *params*, *heap-size*)))
≮ *x*)

THEOREM: lr-eval-s-eval-flag-run-s->lr
let *s-lr* **be** s->lr (*s*, *heap-size*, *max-ctrl*, *max-temp*, *word-size*)
in
(proper-p-statep (lr->p (*s-lr*))
∧ lr-programs-properp (*s-lr*,
cdr (lr-data-seg-table (s-progs (*s*),
s-params (*s*),
heap-size)))
∧ lr-s-similar-statesp (s-params (*s*),
s-temps (*s*),
s-lr,
cdr (lr-data-seg-table (s-progs (*s*),

s-params (*s*),
heap-size)))
∧ s-restrictedp (s-progs (*s*), s-params (*s*))
∧ s-good-statep (*s*, *c*)
∧ s-all-temps-setp (**t**,
s-body (car (s-progs (*s*))),
temp-alist-to-set (s-temps (*s*)))
∧ s-all-progs-temps-setp (s-progs (*s*))
∧ s-check-temps-setp (s-temps (*s*))
∧ (s-err-flag (s-eval (**t**, *s*, *c*)) = 'run)
∧ (*heap-size* ≮ (s-total-heap-reqs (s-progs (*s*),
s-params (*s*),
heap-size)
+ s-eval-heap-r (**t**, *s*, *c*)))
∧ (*max-ctrl* ≮ (p-ctrl-stk-size (p-ctrl-stk (*s-lr*))
+ s-eval-ctrl-r (**t**, *s*, *c*)))
∧ (*max-temp* ≮ s-eval-temp-r (**t**, *s*, *c*))
∧ (*word-size* ≮ s-eval-ws-r (**t**, *s*, *c*))
∧ (*word-size* ≮ S-MAX-SUBR-REQS)
∧ (s-pname (*s*) = name (car (s-progs (*s*))))
∧ (s-pos (*s*) = **nil**))
→ (p-psw (lr-eval (**t**, *s-lr*, *c*)) = 'run) **endlet**

THEOREM: all-undef-addr-strip-cdrs-lr-make-initial-temps
all-undef-addrs (strip-cdrs (lr-make-initial-temps (*x*)))

THEOREM: lr-s-similar-temps-make-temps-entries-initial-temps
(all-undef-addrs (strip-cdrs (*lr-temps*))
∧ (length (*s-temps*) = length (*lr-temps*)))
→ lr-s-similar-temps (make-temps-entries (*s-temps*), *lr-temps*, *data-seg*)

DEFINITION:
object-addrs (*object-list*, *table*)
= **if** listp (*object-list*)
then cons (cdr (assoc (car (*object-list*), *table*)),
object-addrs (cdr (*object-list*), *table*))
else nil endif

THEOREM: lr-valp-lr-compile-quote
(lr-proper-p-areasp (*data-seg*)
∧ lr-s-similar-const-table (*table*, *data-seg*)
∧ s-restricted-objectp (*flag*, *object*)
∧ lr-proper-heapp (*data-seg*)
∧ definedp (**f**, *table*)
∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),

lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs (*flag*, *object*, *data-seg*, *table*)))
$\rightarrow$ **if** *flag* = `'list`
then lr-check-result1 (*object*,
object-addrs (*object*,
cdr (lr-compile-quote (*flag*,
object,
data-seg,
table))),
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*)))
else lr-valp (*object*,
cdr (assoc (*object*,
cdr (lr-compile-quote (*flag*,
object,
data-seg,
table)))),
car (lr-compile-quote (*flag*, *object*, *data-seg*, *table*))) **endif**

THEOREM: lr-init-data-seg-table-preserves-lr-valp
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-init-data-seg-restrictedp (*params*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-init-heap-reqs (*params*, *data-seg*, *table*))
$\wedge$ lr-valp (*value*, *addr*, *data-seg*))
$\rightarrow$ lr-valp (*value*, *addr*, car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))

THEOREM: lr-valp-lr-compile-quote-flag-t
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ s-restricted-objectp (**t**, *object*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)

$\not<$ s-heap-reqs (**t**, *object*, *data-seg*, *table*)))
$\rightarrow$ lr-valp (*object*,
cdr (assoc (*object*,
cdr (lr-compile-quote (**t**, *object*, *data-seg*, *table*)))),
car (lr-compile-quote (**t**, *object*, *data-seg*, *table*)))

THEOREM: lr-s-similar-params-pair-formals-lr-init-data-seg
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-s-similar-const-table (*table*, *data-seg*)
$\wedge$ s-init-data-seg-restrictedp (*params*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-init-heap-reqs (*params*, *data-seg*, *table*)))
$\rightarrow$ lr-s-similar-params (*params*,
pair-formals-with-addresses (*params*,
cdr (lr-init-data-seg-table (*params*,
data-seg,
table))),
car (lr-init-data-seg-table (*params*, *data-seg*, *table*)))

THEOREM: lr-data-seg-table-body-preserves-lr-valp
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-data-seg-body-restrictedp (*flag*, *body*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-body (*flag*, *body*, *data-seg*, *table*))
$\wedge$ lr-valp (*value*, *addr*, *data-seg*))
$\rightarrow$ lr-valp (*value*,
addr,
car (lr-data-seg-table-body (*flag*, *body*, *data-seg*, *table*)))

THEOREM: lr-data-seg-table-list-preserves-lr-valp
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-data-seg-list-restrictedp (*progs*)
$\wedge$ lr-proper-heapp (*data-seg*)

$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-list (*progs*, *data-seg*, *table*))
$\wedge$ lr-valp (*value*, *addr*, *data-seg*))
$\rightarrow$ lr-valp (*value*, *addr*, car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))

THEOREM: lr-data-seg-table-list-preserves-lr-s-similar-params
(lr-proper-p-areasp (*data-seg*)
$\wedge$ lr-good-pointerp-tablep (*table*, *data-seg*)
$\wedge$ s-data-seg-list-restrictedp (*progs*)
$\wedge$ lr-proper-heapp (*data-seg*)
$\wedge$ definedp (**f**, *table*)
$\wedge$ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
$\not<$ s-heap-reqs-list (*progs*, *data-seg*, *table*))
$\wedge$ lr-s-similar-params (*s-params*, *lr-params*, *data-seg*))
$\rightarrow$ lr-s-similar-params (*s-params*,
lr-params,
car (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))

DEFINITION:
all-definedp (*list*, *alist*)
= **if** listp (*list*)
then definedp (car (*list*), *alist*) $\wedge$ all-definedp (cdr (*list*), *alist*)
else t endif

THEOREM: assoc-definedp-table-lr-data-seg-table-body
definedp (*object*, *table*)
$\rightarrow$ (assoc (*object*, cdr (lr-data-seg-table-body (*flag*, *expr*, *data-seg*, *table*)))
= assoc (*object*, *table*))

THEOREM: assoc-definedp-table-lr-data-seg-table-list
definedp (*object*, *table*)
$\rightarrow$ (assoc (*object*, cdr (lr-data-seg-table-list (*progs*, *data-seg*, *table*)))
= assoc (*object*, *table*))

THEOREM: pair-formals-with-addresses-lr-data-seg-table-list
all-definedp (strip-cdrs (*params*), *table*)
$\rightarrow$ (pair-formals-with-addresses (*params*,
cdr (lr-data-seg-table-list (*progs*,

data-seg,
table)))
= pair-formals-with-addresses (*params*, *table*))

THEOREM: all-definedp-strip-cdrs-lr-init-data-seg-table
(lr-s-similar-const-table (*table*, *data-seg*)
∧ lr-proper-p-areasp (*data-seg*)
∧ lr-proper-heapp (*data-seg*)
∧ definedp (**f**, *table*)
∧ s-init-data-seg-restrictedp (*params*)
∧ (lr-count-free-nodes (fetch (LR-FP-ADDR, *data-seg*),
lr-free-list-nodes (lr-max-node (*data-seg*),
data-seg),
data-seg)
≮ s-init-heap-reqs (*params*, *data-seg*, *table*)))
→ all-definedp (strip-cdrs (*params*),
cdr (lr-init-data-seg-table (*params*, *data-seg*, *table*)))

THEOREM: lr-s-similar-params-pair-formals-with-addresses
((*heap-size* ≮ s-total-heap-reqs (*progs*, *params*, *heap-size*))
∧ s-restrictedp (*progs*, *params*))
→ lr-s-similar-params (*params*,
pair-formals-with-addresses (*params*,
cdr (lr-data-seg-table (*progs*,
params,
heap-size))),
car (lr-data-seg-table (*progs*, *params*, *heap-size*)))

THEOREM: lr-s-similar-statesp-s->lr-lr-data-seg-table
((*heap-size* ≮ s-total-heap-reqs (s-progs (*s*), s-params (*s*), *heap-size*))
∧ s-restrictedp (s-progs (*s*), s-params (*s*))
∧ (strip-cars (s-params (*s*)) = s-formals (car (s-progs (*s*))))
∧ (s-pname (*s*) = name (car (s-progs (*s*))))
∧ (s-temps (*s*) = make-temps-entries (s-temp-list (car (s-progs (*s*))))))
→ lr-s-similar-statesp (s-params (*s*),
s-temps (*s*),
s->lr (*s*, *heap-size*, *max-ctrl*, *max-temp*, *word-size*),
cdr (lr-data-seg-table (s-progs (*s*),
s-params (*s*),
heap-size)))

THEOREM: p-ctrl-stk-size-p-ctrl-stk-s->lr
p-ctrl-stk-size (p-ctrl-stk (s->lr (*s*, *heap-size*, *max-ctrl*, *max-temp*, *word-size*)))
= (2 + length (s-params (*s*)) + length (s-temps (*s*)))

```
THEOREM: s->lr-ok
 let s-lr  be s->lr (s, heap-size, max-ctrl, max-temp, word-size)
 in
 (s-good-statep (s, c)
  ∧   s-all-temps-setp (t,
                        s-body (car (s-progs (s))),
                        temp-alist-to-set (s-temps (s)))
  ∧   s-all-progs-temps-setp (s-progs (s))
  ∧   (s-temps (s)
       =   make-temps-entries (s-temp-list (car (s-progs (s)))))
  ∧   s-check-temps-setp (s-temps (s))
  ∧   s-restrictedp (s-progs (s), s-params (s))
  ∧   (heap-size ≮ (s-total-heap-reqs (s-progs (s),
                                       s-params (s),
                                       heap-size)
                    +   s-eval-heap-r (t, s, c)))
  ∧   s-restrict-subrps-progs (s-progs (s))
  ∧   (max-ctrl ≮ (2
                   +   length (s-params (s))
                   +   length (s-temps (s))
                   +   s-eval-ctrl-r (t, s, c)))
  ∧   (max-ctrl ∈ N)
  ∧   (max-ctrl < exp (2, word-size))
  ∧   (max-temp ∈ N)
  ∧   (max-temp ≮ s-eval-temp-r (t, s, c))
  ∧   (max-temp < exp (2, word-size))
  ∧   (word-size ≮ max (s-total-ws-reqs (s-progs (s),
                                         s-params (s),
                                         heap-size),
                        s-eval-ws-r (t, s, c)))
  ∧   (word-size ∈ N)
  ∧   litatom (name (car (s-progs (s))))
  ∧   (s-pname (s) = name (car (s-progs (s))))
  ∧   (s-pos (s) = nil)
  ∧   all-litatoms (strip-cars (s-params (s)))
  ∧   no-duplicatesp (strip-cars (s-progs (s)))
  ∧   (strip-cars (s-params (s)) = s-formals (car (s-progs (s))))
  ∧   (s-err-flag (s-eval (t, s, c)) = 'run))
  →   lr-valp (s-ans (s-eval (t, s, c)),
               car (p-temp-stk (lr-eval (t, s-lr, c))),
               p-data-segment (lr-eval (t, s-lr, c))) endlet

THEOREM: s-good-state-logic->s
 (l-proper-programsp (prog-names)
```

$\wedge$ l-proper-exprp (**t**, *expr*, *prog-names*, strip-cars (*alist*))
$\wedge$ all-litatoms (strip-cars (*alist*)))
$\rightarrow$ s-good-statep (logic->s (*expr*, *alist*, *prog-names*), *c*)

THEOREM: s-body-car-s-progs-logic->s
s-body (car (s-progs (logic->s (*expr*, *alist*, *pnames*)))) = *expr*

THEOREM: l-proper-programsp-s-progs-logic->s
(l-proper-programsp (*pnames*) $\wedge$ l-proper-exprp (**t**, *expr*, *pnames*, *formals*))
$\rightarrow$ s-all-progs-temps-setp (s-progs (logic->s (*expr*, *alist*, *pnames*)))

THEOREM: s-temps-logic->s
s-temps (logic->s (*expr*, *alist*, *pnames*)) = **nil**

THEOREM: s-temp-list-car-s-progs-logic->s
s-temp-list (car (s-progs (logic->s (*expr*, *alist*, *pnames*)))) = **nil**

THEOREM: s-params-logic->s
s-params (logic->s (*expr*, *alist*, *pnames*)) = *alist*

DEFINITION:
l-data-seg-body-restrictedp (*flag*, *expr*)
= **if** *flag* = 'list
then if listp (*expr*)
then l-data-seg-body-restrictedp (**t**, car (*expr*))
$\wedge$ l-data-seg-body-restrictedp ('list, cdr (*expr*))
else t endif
elseif listp (*expr*)
then if car (*expr*) = 'quote
then s-restricted-objectp (**t**, cadr (*expr*))
else l-data-seg-body-restrictedp ('list, cdr (*expr*)) **endif**
else t endif

DEFINITION:
l-data-seg-list-restrictedp (*fun-names*)
= **if** listp (*fun-names*)
then l-data-seg-body-restrictedp (**t**, body (car (*fun-names*)))
$\wedge$ l-data-seg-list-restrictedp (cdr (*fun-names*))
else t endif

DEFINITION:
l-restrictedp (*fun-names*, *alist*)
= (s-init-data-seg-restrictedp (*alist*)
$\wedge$ l-data-seg-list-restrictedp (*fun-names*))

THEOREM: l-data-seg-body-restrictedp-s-data-seg-body-restrictedp
l-data-seg-body-restrictedp (*flag*, *body*)
→ s-data-seg-body-restrictedp (*flag*, *body*)

THEOREM: l-data-seg-body-restrictedp-delete-all
l-data-seg-list-restrictedp (*pnames*)
→ l-data-seg-list-restrictedp (delete-all (*name*, *pnames*))

THEOREM: l-data-seg-list-restrictedp-s-data-seg-list-restrictedp
(l-data-seg-list-restrictedp (*pnames*) ∧ l-data-seg-body-restrictedp (**t**, *expr*))
→ s-data-seg-list-restrictedp (s-progs (logic->s (*expr*, *alist*, *pnames*)))

THEOREM: l-restrict-subrps-s-restrict-subrps
(l-restrict-subrps (*flag*, *expr*)
∧ l-proper-exprp (*flag*, *expr*, *program-names*, *formals*))
→ s-restrict-subrps (*flag*, *expr*)

THEOREM: l-restrict-subrps-progs-delete-all
l-restrict-subrps-progs (*pnames*)
→ l-restrict-subrps-progs (delete-all (*name*, *pnames*))

THEOREM: l-restrict-subrps-progs-s-restrict-subrps-progs
(l-restrict-subrps-progs (*pnames*)
∧ l-proper-programsp-1 (*pnames*, *program-names*))
→ s-restrict-subrps-progs (s-construct-programs (remove-duplicates (*pnames*)))

THEOREM: l-restrict-subrps-progs-s-restrict-subrps-progs-logic->s
(l-restrict-subrps-progs (*pnames*)
∧ l-restrict-subrps (**t**, *expr*)
∧ l-proper-exprp (**t**, *expr*, *program-names*, *formals*)
∧ l-proper-programsp (*pnames*))
→ s-restrict-subrps-progs (s-progs (logic->s (*expr*, *alist*, *pnames*)))

THEOREM: name-car-s-progs-logic->s
name (car (s-progs (logic->s (*expr*, *alist*, *pnames*)))) = `'main`

THEOREM: s-pname-logic->s
s-pname (logic->s (*expr*, *alist*, *pnames*)) = `'main`

THEOREM: s-pos-logic->s
s-pos (logic->s (*expr*, *alist*, *pnames*)) = **nil**

THEOREM: definedp-user-fname-s-construct-programs
(litatom (*name*) ∧ all-litatoms-not-plist (*pnames*))
→ (definedp (user-fname (*name*), s-construct-programs (*pnames*))
= (*name* ∈ *pnames*))

THEOREM: no-duplicatesp-strip-cars-s-construct-programs
all-litatoms-not-plist (*pnames*)
→ (no-duplicatesp (strip-cars (s-construct-programs (*pnames*)))
= no-duplicatesp (*pnames*))

THEOREM: all-user-fnamesp-strip-cars-s-construct-programs
all-litatoms-not-plist (*pnames*)
→ all-user-fnamesp (strip-cars (s-construct-programs (*pnames*)))

THEOREM: no-duplicatesp-strip-cars-s-progs-logic->s
all-litatoms-not-plist (*pnames*)
→ no-duplicatesp (strip-cars (s-progs (logic->s (*expr*, *alist*, *pnames*))))

THEOREM: s-formals-car-s-progs-logic->s
s-formals (car (s-progs (logic->s (*expr*, *alist*, *pnames*)))) = strip-cars (*alist*)

THEOREM: s-expr-logic->s
s-expr (logic->s (*expr*, *alist*, *pnames*)) = *expr*

THEOREM: all-litatoms-not-plist-lr-proper-programsp
l-proper-programsp (*prog-names*) → all-litatoms-not-plist (*prog-names*)

THEOREM: logic->lr-ok-really
(l-proper-exprp (**t**, *expr*, *pnames*, strip-cars (*alist*))
∧ l-proper-programsp (*pnames*)
∧ all-litatoms (strip-cars (*alist*))
∧ l-data-seg-body-restrictedp (**t**, *expr*)
∧ l-restrictedp (*pnames*, *alist*)
∧ v&c$ (**t**, *expr*, *alist*)
∧ (cdr (v&c$ (**t**, *expr*, *alist*)) < *c*)
∧ l-restrict-subrps (**t**, *expr*)
∧ l-restrict-subrps-progs (*pnames*)
∧ (*heap-size* ≮ lr-total-heap-reqs (*expr*, *alist*, *pnames*, *heap-size*, *c*))
∧ (*max-ctrl* ≮ lr-max-ctrl-reqs (*expr*, *alist*, *pnames*, *c*))
∧ (*max-ctrl* < exp (2, *word-size*))
∧ (*max-ctrl* ∈ **N**)
∧ (*max-temp* ≮ lr-max-temp-reqs (*expr*, *alist*, *pnames*, *c*))
∧ (*max-temp* < exp (2, *word-size*))
∧ (*max-temp* ∈ **N**)
∧ (*word-size* ≮ lr-max-word-size-reqs (*expr*, *alist*, *pnames*, *heap-size*, *c*))
∧ (*word-size* ∈ **N**))
→ lr-valp (car (v&c$ (**t**, *expr*, *alist*)),
car (p-temp-stk (lr-eval (**t**,
s->lr (logic->s (*expr*, *alist*, *pnames*),
heap-size,

max-ctrl,
max-temp,
word-size),
c))),
p-data-segment (lr-eval (**t**,
s->lr (logic->s (*expr*, *alist*, *pnames*),
heap-size,
max-ctrl,
max-temp,
word-size),
c)))

```
; -------------------------------------------------------------
; was p1.events
; -------------------------------------------------------------
```

DEFINITION:
logic->p (*expr*, *alist*, *pnames*, *heap-size*, *max-ctrl*, *max-temp*, *word-size*)
= lr->p (s->lr (logic->s (*expr*, *alist*, *pnames*),
heap-size,
max-ctrl,
max-temp,
word-size))

DEFINITION:
p-run-subr-clock (*l*, *new-l*)
= **case on** car (lr-expr (*l*)):
case = *car*
then p-car-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
case = *cdr*
then p-cdr-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
case = *cons*
then p-cons-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
case = *false*
then p-false-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
case = *falsep*
then p-falsep-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
case = *listp*
then p-listp-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
case = *nlistp*
then p-nlistp-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
case = *true*
then p-true-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
case = *truep*

then p-truep-clock (p-set-pc (lr->p (*new-l*), lr-return-pc (*l*)))
otherwise 0 **endcase**

EVENT: Disable p-run-subr-clock.

DEFINITION:
p-clock1 (*flag*, *l*, *c*)
= **if** p-psw (*l*) ≠ 'run **then** 0
elseif *flag* = 'list
then if offset (p-pc (*l*)) ≃ **nil** **then** 0
elseif listp (lr-expr-list (*l*))
then p-clock1 (**t**, *l*, *c*)
+ p-clock1 ('list,
lr-set-expr (lr-eval (**t**, *l*, *c*),
l,
nx (offset (p-pc (*l*)))),
c)
else 0 **endif**
elseif *c* ≃ 0 **then** 0
elseif litatom (lr-expr (*l*)) **then** 1
elseif lr-expr (*l*) ≃ **nil** **then** 0
elseif car (lr-expr (*l*)) = 'if
then let *test* **be** lr-if-ok (lr-eval (**t**,
lr-set-pos (*l*,
dv (offset (p-pc (*l*)), 1)),
c))
in
if p-psw (*test*) = 'run
then if top (p-temp-stk (*test*)) ≠ LR-F-ADDR
then p-clock1 (**t**,
lr-set-pos (*l*, dv (offset (p-pc (*l*)), 1)),
c)
+ 3
+ p-clock1 (**t**,
lr-set-expr (lr-pop-tstk (*test*),
l,
dv (offset (p-pc (*l*)),
2)),
c)
+ 1
else p-clock1 (**t**,
lr-set-pos (*l*,
dv (offset (p-pc (*l*)), 1)),
c)

```
                + 3
                + p-clock1 (t,
                            lr-set-expr (lr-pop-tstk (test),
                                         l,
                                         dv (offset (p-pc (l)),
                                             3)),
                            c) endif
        else p-clock1 (t,
                       lr-set-pos (l, dv (offset (p-pc (l)), 1)),
                       c) endif endlet
elseif car (lr-expr (l)) = S-TEMP-EVAL
then p-clock1 (t, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c) + 1
elseif car (lr-expr (l)) = S-TEMP-TEST
then if lr-eval-temp-setp (l) then 5
        else 4
             + p-clock1 (t,
                         lr-set-pos (l, dv (offset (p-pc (l)), 1)),
                         c)
             + 2 endif
elseif car (lr-expr (l)) = S-TEMP-FETCH then 1
elseif car (lr-expr (l)) = 'quote then 1
elseif p-psw (lr-eval ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c))
        ≠ 'run
then p-clock1 ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
elseif subrp (car (lr-expr (l)))
then p-clock1 ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
        + p-run-subr-clock (l,
                            lr-eval ('list,
                                     lr-set-pos (l,
                                                 dv (offset (p-pc (l)), 1)),
                                     c))
elseif litatom (car (lr-expr (l)))
then let fs be lr-funcall (l,
                           lr-eval ('list,
                                    lr-set-pos (l,
                                                dv (offset (p-pc (l)), 1)),
                                    c))
        in
        p-clock1 ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
        + 1
        + p-clock1 (t, fs, c − 1)
        + 1 endlet
else 0 endif
```

DEFINITION:
lr-good-pointerps (*list*, *data-seg*)
= **if** listp (*list*)
then lr-good-pointerp (car (*list*), *data-seg*)
∧ lr-good-pointerps (cdr (*list*), *data-seg*)
else t endif

DEFINITION:
lr-proper-ctrl-stkp (*ctrl-stk*, *data-seg*)
= **if** *ctrl-stk* ≃ **nil** **then** *ctrl-stk* = **nil**
else lr-good-pointerps (strip-cdrs (bindings (top (*ctrl-stk*))),
data-seg)
∧ lr-proper-ctrl-stkp (pop (*ctrl-stk*), *data-seg*) **endif**

DEFINITION:
lr-p-proper-statep (*temp-stk*, *ctrl-stk*, *data-seg*, *table*)
= (lr-proper-ctrl-stkp (*ctrl-stk*, *data-seg*)
∧ lr-good-pointerps (*temp-stk*, *data-seg*)
∧ lr-proper-heapp (*data-seg*)
∧ lr-good-pointerp-tablep (*table*, *data-seg*))

EVENT: Disable lr-p-proper-statep.

THEOREM: definedp-cdr-assoc-lr-good-pointerps
((*addr* ∈ *list*) ∧ lr-good-pointerps (*list*, *data-seg*))
→ lr-good-pointerp (*addr*, *data-seg*)

THEOREM: lr-p-proper-statep-lr-push-tstk-cdr-assoc-lr-expr
(litatom (lr-expr (*l*))
∧ proper-p-statep (lr->p (*l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ lr-p-proper-statep (p-temp-stk (*l*),
p-ctrl-stk (*l*),
p-data-segment (*l*),
table)
∧ (p-psw (lr-push-tstk (*l*,
cdr (assoc (lr-expr (*l*),
bindings (car (p-ctrl-stk (*l*)))))))
= 'run))
→ lr-p-proper-statep (p-temp-stk (lr-push-tstk (*l*,
cdr (assoc (lr-expr (*l*),
bindings (car (p-ctrl-stk (*l*)))))))),
p-ctrl-stk (*l*),

p-data-segment (*l*),
table)

THEOREM: lr-p-proper-statep-cdr-temp-stk
lr-p-proper-statep (*temp-stk*, *ctrl-stk*, *data-seg*, *table*)
→ lr-p-proper-statep (cdr (*temp-stk*), *ctrl-stk*, *data-seg*, *table*)

THEOREM: lr-good-pointerps-put-assoc
(lr-good-pointerp (*addr*, *data-seg*)
∧ lr-good-pointerps (strip-cdrs (*bindings*), *data-seg*))
→ lr-good-pointerps (strip-cdrs (put-assoc (*addr*, *expr*, *bindings*)), *data-seg*)

THEOREM: lr-p-proper-statep-cons-p-frame-put-assoc
(lr-p-proper-statep (*temp-stk*, *ctrl-stk*, *data-seg*, *table*)
∧ listp (*temp-stk*)
∧ listp (*ctrl-stk*)
∧ (*cdr-ctrl-stk* = cdr (*ctrl-stk*)))
→ lr-p-proper-statep (*temp-stk*,
cons (p-frame (put-assoc (car (*temp-stk*),
expr,
bindings (car (*ctrl-stk*))),
ret-pc),
cdr-ctrl-stk),
data-seg,
table)

THEOREM: lr-eval-leaves-listp-p-ctrl-stk-lr->p-lr-set-pos
(proper-p-statep (lr->p (*l*))
∧ good-posp (*flag*, *pos*, program-body (p-current-program (*l*)))
∧ good-posp (*flag*, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (p-psw (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)) = `'run`))
→ listp (p-ctrl-stk (lr-eval (*flag*, lr-set-pos (*l*, *pos*), *c*)))

THEOREM: lr-p-proper-statep-cdr-assoc-caddr-lr-expr-bindings
(proper-p-statep (lr->p (*l*))
∧ lr-p-proper-statep (*temp-stk*, p-ctrl-stk (*l*), *data-seg*, *table*)
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ listp (lr-expr (*l*))
∧ ((car (lr-expr (*l*)) = S-TEMP-TEST)
∨ (car (lr-expr (*l*)) = S-TEMP-FETCH)))
→ lr-p-proper-statep (cons (cdr (assoc (caddr (lr-expr (*l*)),
bindings (car (p-ctrl-stk (*l*))))),
temp-stk),

p-ctrl-stk (l),
data-seg,
table)

THEOREM: member-strip-cdrs-lr-good-pointerp-tablep
((*object* ∈ strip-cdrs (*table*)) ∧ lr-good-pointerp-tablep (*table*, *data-seg*))
→ lr-good-pointerp (*object*, *data-seg*)

THEOREM: lr-p-proper-statep-p-temps-stk-lr-push-tstk-quote
(listp (lr-expr (l))
∧ (car (lr-expr (l)) = `'quote`)
∧ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
∧ lr-programs-properp (l, *table*)
∧ (p-psw (lr-push-tstk (l, cadr (lr-expr (l)))) = `'run`)
∧ lr-p-proper-statep (p-temp-stk (l), *ctrl-stk*, *data-seg*, *table*))
→ lr-p-proper-statep (p-temp-stk (lr-push-tstk (l, cadr (lr-expr (l)))),
ctrl-stk,
data-seg,
table)

THEOREM: p-run-subr-preserves-lr-good-pointerp-tablep
(listp (lr-expr (l))
∧ (car (lr-expr (l)) ≠ `'if`)
∧ (car (lr-expr (l)) ≠ `'quote`)
∧ (p-psw (*new-l*) = `'run`)
∧ subrp (car (lr-expr (l)))
∧ lr-good-pointerp-tablep (*table2*, p-data-segment (*new-l*))
∧ proper-p-statep (lr->p (l))
∧ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
∧ lr-programs-properp (l, *table1*)
∧ lr-programs-properp (*new-l*, *table1*)
∧ (p-psw (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p (*new-l*), lr-return-pc (l))))
= `'run`)
∧ lr-proper-heapp (p-data-segment (*new-l*))
∧ proper-p-statep (lr->p (*new-l*))
∧ (p-prog-segment (*new-l*) = p-prog-segment (l)))
→ lr-good-pointerp-tablep (*table2*,
p-data-segment (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p (*new-l*),
lr-return-pc (l)))))

THEOREM: lr-good-pointerps-deposit-free-ptr
lr-good-pointerps (*list*, deposit (*anything*, identity (LR-FP-ADDR), *data-seg*))
= lr-good-pointerps (*list*, *data-seg*)

THEOREM: lr-proper-ctrl-stkp-deposit-free-ptr
lr-proper-ctrl-stkp (*ctrl-stk*, deposit (*anything*, identity (LR-FP-ADDR), *data-seg*))
= lr-proper-ctrl-stkp (*ctrl-stk*, *data-seg*)

THEOREM: lr-good-pointerp-deposit-a-list-node
(lr-good-pointerp (*addr1*, *data-seg*)
∧ (type (*addr2*) = 'addr)
∧ (cddr (*addr2*) = **nil**)
∧ listp (*addr2*)
∧ adpp (untag (*addr2*), *data-seg*)
∧ lr-boundary-nodep (*addr2*)
∧ (area-name (*addr2*) = 'heap)
∧ (type (*ref-count*) = 'nat))
→ lr-good-pointerp (*addr1*,
deposit-a-list (list (*x*, *ref-count*, *y*, *z*), *addr2*, *data-seg*))

THEOREM: lr-good-pointerps-deposit-a-list-node
(lr-good-pointerps (*list*, *data-seg*)
∧ (type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ adpp (untag (*addr*), *data-seg*)
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = 'heap)
∧ (type (*ref-count*) = 'nat))
→ lr-good-pointerps (*list*,
deposit-a-list (list (*x*, *ref-count*, *y*, *z*), *addr*, *data-seg*))

THEOREM: lr-proper-ctrl-stkp-deposit-a-list-node
(lr-proper-ctrl-stkp (*list*, *data-seg*)
∧ (type (*addr*) = 'addr)
∧ (cddr (*addr*) = **nil**)
∧ listp (*addr*)
∧ adpp (untag (*addr*), *data-seg*)
∧ lr-boundary-nodep (*addr*)
∧ (area-name (*addr*) = 'heap)
∧ (type (*ref-count*) = 'nat))
→ lr-proper-ctrl-stkp (*list*,
deposit-a-list (list (*x*, *ref-count*, *y*, *z*),
addr,
data-seg))

THEOREM: p-run-subr-preserves-lr-proper-ctrl-stkp
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)

$\wedge$ listp (lr-expr (l))
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ lr-proper-heapp (p-data-segment (lr-eval ('list, lr-set-pos (l, pos), c)))
$\wedge$ lr-proper-ctrl-stkp (p-ctrl-stk (lr-eval ('list, lr-set-pos (l, pos), c)),
p-data-segment (lr-eval ('list,
lr-set-pos (l, pos),
c)))
$\wedge$ (p-psw (lr-eval ('list, lr-set-pos (l, pos), c)) = 'run)
$\wedge$ (p-psw (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p (lr-eval ('list, lr-set-pos (l, pos), c)),
lr-return-pc (l))))
= 'run)
$\wedge$ (pos = dv (offset (p-pc (l)), 1)))
$\rightarrow$ lr-proper-ctrl-stkp (p-ctrl-stk (lr-eval ('list, lr-set-pos (l, pos), c)),
p-data-segment (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p (lr-eval ('list,
lr-set-pos (l,
pos),
c)),
lr-return-pc (l)))))

THEOREM: lr-good-pointerps-cons-lr-f-addr-lr-proper-heapp
(lr-good-pointerps ($list$, $data\text{-}seg$)
$\wedge$ lr-proper-heapp ($data\text{-}seg$)
$\wedge$ lr-proper-p-areasp ($data\text{-}seg$))
$\rightarrow$ lr-good-pointerps (cons (identity (LR-F-ADDR), $list$), $data\text{-}seg$)

THEOREM: lr-good-pointerps-cons-lr-t-addr-lr-proper-heapp
(lr-good-pointerps ($list$, $data\text{-}seg$)
$\wedge$ lr-proper-heapp ($data\text{-}seg$)
$\wedge$ lr-proper-p-areasp ($data\text{-}seg$))
$\rightarrow$ lr-good-pointerps (cons (identity (LR-T-ADDR), $list$), $data\text{-}seg$)

THEOREM: lr-good-pointerps-cons-lr-0-addr-lr-proper-heapp
(lr-good-pointerps ($list$, $data\text{-}seg$)
$\wedge$ lr-proper-heapp ($data\text{-}seg$)
$\wedge$ lr-proper-p-areasp ($data\text{-}seg$))
$\rightarrow$ lr-good-pointerps (cons (identity (LR-0-ADDR), $list$), $data\text{-}seg$)

THEOREM: lr-good-pointerps-cdr
lr-good-pointerps ($list$, $data\text{-}seg$) $\rightarrow$ lr-good-pointerps (cdr ($list$), $data\text{-}seg$)

THEOREM: lr-good-pointerps-cons-fetch-car-temp-stk-cdr-car
(lr-good-pointerps (*temp-stk*, *data-seg*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ (length (*temp-stk*) ≮ `1`)
 ∧ (fetch (car (*temp-stk*), *data-seg*) = tag (`'nat`, LR-CONS-TAG)))
→ lr-good-pointerps (cons (fetch (add-addr (car (*temp-stk*),
identity (LR-CAR-OFFSET)),
data-seg),
cdr (*temp-stk*)),
data-seg)

THEOREM: lr-good-pointerps-cons-fetch-car-temp-stk-cdr-cdr
(lr-good-pointerps (*temp-stk*, *data-seg*)
 ∧ lr-proper-heapp (*data-seg*)
 ∧ lr-proper-p-areasp (*data-seg*)
 ∧ (length (*temp-stk*) ≮ `1`)
 ∧ (fetch (car (*temp-stk*), *data-seg*) = tag (`'nat`, LR-CONS-TAG)))
→ lr-good-pointerps (cons (fetch (add-addr (car (*temp-stk*),
identity (LR-CDR-OFFSET)),
data-seg),
cdr (*temp-stk*)),
data-seg)

THEOREM: lr-good-pointerps-cons-fetch-fp-addr-deposit-a-list-cons
(lr-proper-heapp (*data-seg*)
 ∧ lr-good-pointerps (*temp-stk*, *data-seg*)
 ∧ (*fp-addr* = fetch (identity (LR-FP-ADDR), *data-seg*))
 ∧ (type (*ref-count*) = `'nat`))
→ lr-good-pointerps (cons (fetch (identity (LR-FP-ADDR), *data-seg*), *temp-stk*),
deposit-a-list (list (*x*, *ref-count*, *y*, *z*),
fp-addr,
data-seg))

THEOREM: p-run-subr-preserves-lr-good-pointerps
let *lr-eval* **be** lr-eval (`'list`, lr-set-pos (*l*, *pos*), *c*)
in
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ listp (lr-expr (*l*))
 ∧ subrp (car (lr-expr (*l*)))
 ∧ (car (lr-expr (*l*)) ≠ `'if`)
 ∧ (car (lr-expr (*l*)) ≠ `'quote`)
 ∧ proper-p-statep (lr->p (*l*))
 ∧ lr-proper-heapp (p-data-segment (*lr-eval*))

$\wedge$ lr-good-pointerps (p-temp-stk (*lr-eval*), p-data-segment (*lr-eval*))
$\wedge$ (p-psw (*lr-eval*) = `'run`)
$\wedge$ (p-psw (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*lr-eval*), lr-return-pc (*l*))))
= `'run`)
$\wedge$ (length (p-temp-stk (*lr-eval*)) $\not<$ arity (car (lr-expr (*l*))))
$\wedge$ (*pos* = dv (offset (p-pc (*l*)), `1`)))
$\rightarrow$ lr-good-pointerps (p-temp-stk (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*lr-eval*),
lr-return-pc (*l*)))),
p-data-segment (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*lr-eval*),
lr-return-pc (*l*))))) **endlet**

THEOREM: p-run-subr-preserves-lr-proper-heapp2-alt
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ lr-programs-properp (*new-l*, *table*)
$\wedge$ listp (lr-expr (*l*))
$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ lr-proper-free-listp (p-data-segment (*new-l*))
$\wedge$ adpp (untag (lr-max-node (p-data-segment (*new-l*))), p-data-segment (*new-l*))
$\wedge$ lr-boundary-nodep (lr-max-node (p-data-segment (*new-l*)))
$\wedge$ (p-psw (*new-l*) = `'run`)
$\wedge$ (p-psw (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*), lr-return-pc (*l*))))
= `'run`)
$\wedge$ lr-good-pointerps (p-temp-stk (*new-l*), p-data-segment (*new-l*))
$\wedge$ (length (p-temp-stk (*new-l*)) $\not<$ arity (car (lr-expr (*l*))))
$\wedge$ lr-proper-heapp2 (*addr*, p-data-segment (*new-l*))
$\wedge$ lr-nodep (*addr*, p-data-segment (*new-l*))
$\wedge$ (p-prog-segment (*l*) = p-prog-segment (*new-l*)))
$\rightarrow$ lr-proper-heapp2 (*addr*,
p-data-segment (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*),
lr-return-pc (*l*)))))

THEOREM: p-run-subr-preserves-lr-proper-heapp
(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ lr-programs-properp (*new-l*, *table*)
$\wedge$ listp (lr-expr (*l*))
$\wedge$ (car (lr-expr (*l*)) $\neq$ `'if`)
$\wedge$ subrp (car (lr-expr (*l*)))

$\wedge$ proper-p-statep (lr->p (*new-l*))
$\wedge$ (p-psw (*new-l*) = 'run)
$\wedge$ (p-psw (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*)), lr-return-pc (*l*))))
= 'run)
$\wedge$ (p-prog-segment (*l*) = p-prog-segment (*new-l*))
$\wedge$ (area-name (p-pc (*l*)) = area-name (p-pc (*new-l*)))
$\wedge$ lr-proper-heapp (p-data-segment (*new-l*))
$\wedge$ lr-good-pointerps (p-temp-stk (*new-l*), p-data-segment (*new-l*))
$\wedge$ (length (p-temp-stk (*new-l*)) $\not<$ arity (car (lr-expr (*l*)))))
$\rightarrow$ lr-proper-heapp (p-data-segment (p-run-subr (car (lr-expr (*l*)),
p-set-pc (lr->p (*new-l*),
lr-return-pc (*l*)))))

THEOREM: lr-apply-subr-preserves-lr-p-proper-statep
let *lr-eval* **be** lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)
in
(listp (lr-expr (*l*))
$\wedge$ (car (lr-expr (*l*)) $\neq$ 'if)
$\wedge$ (car (lr-expr (*l*)) $\neq$ 'quote)
$\wedge$ (car (lr-expr (*l*)) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (lr-expr (*l*)) $\neq$ S-TEMP-TEST)
$\wedge$ (car (lr-expr (*l*)) $\neq$ S-TEMP-FETCH)
$\wedge$ (p-psw (lr-eval ('list, lr-set-pos (*l*, *pos*), *c*)) = 'run)
$\wedge$ subrp (car (lr-expr (*l*)))
$\wedge$ lr-p-proper-statep (p-temp-stk (*lr-eval*),
p-ctrl-stk (*lr-eval*),
p-data-segment (*lr-eval*),
table)
$\wedge$ proper-p-statep (lr->p (*l*))
$\wedge$ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ (p-psw (lr-apply-subr (*l*, *lr-eval*)) = 'run)
$\wedge$ lr-proper-formalsp (cdr (p-prog-segment (*l*)))
$\wedge$ (*pos* = dv (offset (p-pc (*l*)), 1)))
$\rightarrow$ lr-p-proper-statep (p-temp-stk (lr-apply-subr (*l*, *lr-eval*)),
p-ctrl-stk (*lr-eval*),
p-data-segment (lr-apply-subr (*l*, *lr-eval*)),
table) **endlet**

THEOREM: strip-cdrs-append
strip-cdrs (append (*x*, *y*)) = append (strip-cdrs (*x*), strip-cdrs (*y*))

THEOREM: strip-cdrs-pairlist
(length (*x*) $\not<$ length (*y*))

$\rightarrow$ (strip-cdrs (pairlist (x, y)) = first-n (length (x), y))

THEOREM: lr-good-pointerps-append
lr-good-pointerps (append (x, y), *data-seg*)
= (lr-good-pointerps (x, *data-seg*) $\wedge$ lr-good-pointerps (y, *data-seg*))

THEOREM: lr-good-pointerps-reverse
lr-good-pointerps (reverse (x), *data-seg*) = lr-good-pointerps (x, *data-seg*)

THEOREM: lr-good-pointerps-first-n
(lr-good-pointerps (*list*, *data-seg*) $\wedge$ (length (*list*) $\not<$ n))
$\rightarrow$ lr-good-pointerps (first-n (n, *list*), *data-seg*)

DEFINITION:
all-numberps (*list*)
= **if** listp (*list*) **then** (car (*list*) $\in$ **N**) $\wedge$ all-numberps (cdr (*list*))
else t endif

THEOREM: all-numberps-strip-cadrs-numberp-cdr-assoc
all-numberps (strip-cadrs (*list*)) $\rightarrow$ (cadr (assoc (x, *list*)) $\in$ **N**)

THEOREM: all-numberps-strip-cadrs-subr-arity-alist
all-numberps (strip-cadrs (SUBR-ARITY-ALIST))

THEOREM: numberp-arity
arity (x) $\in$ **N**

THEOREM: strip-cdrs-pair-temps-with-initial-values
strip-cdrs (pair-temps-with-initial-values (*temp-var-dcls*))
= strip-cadrs (*temp-var-dcls*)

THEOREM: lr-good-pointerps-all-undef-addrs
(lr-proper-heapp (*data-seg*)
$\wedge$ lr-proper-p-areasp (*data-seg*)
$\wedge$ all-undef-addrs (*list*))
$\rightarrow$ lr-good-pointerps (*list*, *data-seg*)

THEOREM: all-undef-addrs-strip-cadrs-temp-vars-programs-properp-1
(lr-programs-properp-1 (*programs*, *program-names*, *table*)
$\wedge$ definedp (*name*, *programs*))
$\rightarrow$ all-undef-addrs (strip-cadrs (temp-var-dcls (assoc (*name*, *programs*))))

THEOREM: all-undef-addrs-strip-cadrs-temp-vars-programs-properp
(lr-programs-properp (l, *table*) $\wedge$ definedp (*name*, cdr (p-prog-segment (l))))
$\rightarrow$ all-undef-addrs (strip-cadrs (temp-var-dcls (assoc (*name*,
cdr (p-prog-segment (l))))))

THEOREM: lr-good-pointerps-popn
(lr-good-pointerps (*list*, *data-seg*) ∧ (length (*list*) ≮ *n*))
→ lr-good-pointerps (popn (*n*, *list*), *data-seg*)

THEOREM: lr-p-proper-statep-lr-funcall
((p-psw (lr-eval (**t**, lr-funcall (*l*, *new-l*), *c* − 1)) = '**run**)
∧ listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) ≠ '**if**)
∧ (car (lr-expr (*l*)) ≠ '**quote**)
∧ litatom (car (lr-expr (*l*)))
∧ (¬ subrp (car (lr-expr (*l*))))
∧ proper-p-statep (lr->p (*new-l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-p-proper-statep (p-temp-stk (*l*),
p-ctrl-stk (*l*),
p-data-segment (*l*),
table)
∧ lr-p-proper-statep (p-temp-stk (*new-l*),
p-ctrl-stk (*new-l*),
p-data-segment (*new-l*),
table)
∧ (length (p-temp-stk (*new-l*)) ≮ arity (car (lr-expr (*l*))))
∧ lr-proper-formalsp (cdr (p-prog-segment (*l*)))
∧ lr-programs-properp (*l*, *table*))
→ lr-p-proper-statep (p-temp-stk (lr-funcall (*l*, *new-l*)),
p-ctrl-stk (lr-funcall (*l*, *new-l*)),
p-data-segment (*new-l*),
table)

THEOREM: length-p-temp-stk-lr-eval-flag-list-alt
(proper-p-statep (lr->p (*l*))
∧ listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) ≠ '**if**)
∧ (car (lr-expr (*l*)) ≠ '**quote**)
∧ (¬ subrp (car (lr-expr (*l*))))
∧ litatom (car (lr-expr (*l*)))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ lr-proper-formalsp (cdr (p-prog-segment (*l*)))
∧ (p-psw (lr-eval ('**list**, lr-set-pos (*l*, dv (offset (p-pc (*l*)), **1**)), *c*))
= '**run**))
→ (length (p-temp-stk (lr-eval ('**list**,
lr-set-pos (*l*, dv (offset (p-pc (*l*)), **1**)),
c)))
= (arity (car (lr-expr (*l*))) + length (p-temp-stk (*l*))))

THEOREM: lr-p-proper-statep-cdr-lr-ctrl-stk

```
(listp (lr-expr (l))
 ∧  (car (lr-expr (l)) ≠ 'if)
 ∧  (¬ subrp (car (lr-expr (l))))
 ∧  (car (lr-expr (l)) ≠ 'quote)
 ∧  litatom (car (lr-expr (l)))
 ∧  lr-p-proper-statep (temp-stk,
                        p-ctrl-stk (lr-eval (t,
                                             lr-funcall (l,
                                                         lr-eval ('list,
                                                                  lr-set-pos (l,
                                                                              pos),
                                                                  c)),
                                             c − 1)),
                        data-segment,
                        table)
 ∧  good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  lr-programs-properp (l, table)
 ∧  proper-p-statep (lr->p (l))
 ∧  (p-psw (lr-eval (t,
                     lr-funcall (l, lr-eval ('list, lr-set-pos (l, pos), c)),
                     c − 1))
     =  'run)
 ∧  (p-psw (lr-eval ('list, lr-set-pos (l, pos), c)) = 'run)
 ∧  (pos = dv (offset (p-pc (l)), 1)))
→  lr-p-proper-statep (temp-stk,
                        cdr (p-ctrl-stk (lr-eval (t,
                                                  lr-funcall (l,
                                                              lr-eval ('list,
                                                                       lr-set-pos (l,
                                                                                   pos),
                                                                       c)),
                                                  c − 1))),
                        data-segment,
                        table)
```

THEOREM: lr-eval-preserves-lr-p-proper-statep

```
(proper-p-statep (lr->p (l))
 ∧  good-posp (flag, offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  (p-psw (lr-eval (flag, l, c)) = 'run)
 ∧  lr-p-proper-statep (p-temp-stk (l),
                        p-ctrl-stk (l),
                        p-data-segment (l),
                        table)
```

$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ lr-proper-formalsp (cdr (p-prog-segment (l)))
$\rightarrow$ lr-p-proper-statep (p-temp-stk (lr-eval (*flag*, l, c)),
p-ctrl-stk (lr-eval (*flag*, l, c)),
p-data-segment (lr-eval (*flag*, l, c)),
table)

THEOREM: p-plus
p (*p*, *c1* + *c2*) = p (p (*p*, *c1*), *c2*)

THEOREM: p-set-pc-lr->p-lr-set-expr
(p-prog-segment (*l1*) = p-prog-segment (*l2*))
$\rightarrow$ (lr->p (lr-set-expr (*l1*, *l2*, *pos*))
= p-set-pc (lr->p (*l1*), lr-p-pc (lr-set-expr (*l1*, *l2*, *pos*))))

EVENT: Disable p-set-pc-lr->p-lr-set-expr.

THEOREM: member-assoc-area-name-cdr-lr-programs-properp
((assoc (area-name (p-pc (l)), cdr (p-prog-segment (l))) $\notin$ p-prog-segment (l))
$\wedge$ (area-name (p-pc (l)) $\neq$ caar (p-prog-segment (l))))
$\rightarrow$ ($\neg$ lr-programs-properp (l, *table*))

EVENT: Disable member-assoc-area-name-cdr-lr-programs-properp.

THEOREM: not-listp-prog-segment-not-lr-programs-properp
($\neg$ listp (p-prog-segment (l))) $\rightarrow$ ($\neg$ lr-programs-properp (l, *table*))

EVENT: Disable not-listp-prog-segment-not-lr-programs-properp.

THEOREM: unlabel-get-lr-p-pc-program-body-assoc-comp-programs
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*))
$\rightarrow$ (get (lr-p-pc-1 (program-body (p-current-program (l)), offset (p-pc (l))),
program-body (assoc (area-name (p-pc (l)),
comp-programs (p-prog-segment (l)))))
= list (`'dl`,
lr-make-label (offset (lr-p-pc (l))),
nil,
car (comp-body-1 (**t**, lr-expr (l), offset (lr-p-pc (l))))))

THEOREM: car-comp-body-1-litatom
litatom (*body*)
$\rightarrow$ (car (comp-body-1 (**t**, *body*, n)) = list (`'push-local`, *body*))

THEOREM: lr-p-pc-1-append-helper-1
(listp (*body*) ∧ (car (*body*) = 'if) ∧ (*n* ≠ 0))
→ (lr-p-pc-1 (*body*, cons (*n*, *pos*))
= **if** *n* = 1 **then** lr-p-pc-1 (cadr (*body*), *pos*)
elseif *n* = 2
then 3
+ lr-p-c-size (**t**, cadr (*body*))
+ lr-p-pc-1 (caddr (*body*), *pos*)
else lr-p-c-size (**t**, cadr (*body*))
+ lr-p-c-size (**t**, caddr (*body*))
+ lr-p-pc-1 (cadddr (*body*), *pos*)
+ 4 **endif**)

THEOREM: lr-p-pc-1-append-helper-2
(listp (*body*) ∧ (car (*body*) = S-TEMP-EVAL))
→ (lr-p-pc-1 (*body*, cons (1, *pos*)) = lr-p-pc-1 (cadr (*body*), *pos*))

THEOREM: lr-p-pc-1-append-helper-3
(listp (*body*) ∧ (car (*body*) = S-TEMP-TEST))
→ (lr-p-pc-1 (*body*, cons (1, *pos*)) = (lr-p-pc-1 (cadr (*body*), *pos*) + 4))

THEOREM: lr-p-pc-1-append-helper-4
(listp (*body*)
∧ (car (*body*) ≠ S-TEMP-FETCH)
∧ (car (*body*) ≠ S-TEMP-EVAL)
∧ (car (*body*) ≠ S-TEMP-TEST)
∧ (car (*body*) ≠ 'quote)
∧ (car (*body*) ≠ 'if)
∧ (*n* ≠ 0))
→ (lr-p-pc-1 (*body*, cons (*n*, *pos*))
= (lr-p-c-size-list (*n* − 1, *body*) + lr-p-pc-1 (get (*n*, *body*), *pos*)))

THEOREM: lr-p-pc-1-append
(good-posp1 (*pos1*, *body*)
∧ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*))
→ (lr-p-pc-1 (*body*, append (*pos1*, *pos2*))
= (lr-p-pc-1 (*body*, *pos1*) + lr-p-pc-1 (cur-expr (*pos1*, *body*), *pos2*)))

THEOREM: append-butlast-list-car-last
listp (*x*) → (append (butlast (*x*), list (car (last (*x*)))) = plist (*x*))

THEOREM: listp-plist-car
listp (*x*) → (car (plist (*x*)) = car (*x*))

THEOREM: lr-p-pc-1-plist
lr-p-pc-1 (*body*, plist (*pos*)) = lr-p-pc-1 (*body*, *pos*)

THEOREM: lr-p-pc-1-listp-offset
(listp (*pos*)
 ∧ good-posp1 (butlast (*pos*), *body*)
 ∧ lr-proper-exprp (**t**, *body*, *pnames*, *formals*, *temps*, *table*))
→ (lr-p-pc-1 (*body*, *pos*)
  = (lr-p-pc-1 (*body*, butlast (*pos*))
    + lr-p-pc-1 (cur-expr (butlast (*pos*), *body*),
                 list (car (last (*pos*))))))

THEOREM: lr-p-pc-1-nil
lr-p-pc-1 (*body*, **nil**) = `0`

THEOREM: lr-p-pc-1-nx-helper
(listp (*expr*)
 ∧ (car (*expr*) ≠ S-TEMP-FETCH)
 ∧ (car (*expr*) ≠ S-TEMP-EVAL)
 ∧ (car (*expr*) ≠ S-TEMP-TEST)
 ∧ (car (*expr*) ≠ `'quote`)
 ∧ (*n* ≯ `0`)
 ∧ (*n* < length (*expr*)))
→ (lr-p-pc-1 (*expr*, list (*n*))
  = **if** car (*expr*) = `'if`
    then case on *n*:
      case = `1`
      then `0`
      case = `2`
       then `3` + lr-p-c-size (**t**, cadr (*expr*))
      otherwise lr-p-c-size (**t**, cadr (*expr*))
                  + lr-p-c-size (**t**, caddr (*expr*))
                  + `4` **endcase**
    else lr-p-c-size-list (*n* − 1, *expr*) **endif**)

THEOREM: lr-p-pc-1-nx
(lr-programs-properp (*l*, *table*)
 ∧ good-posp (`'list`, offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ listp (offset (p-pc (*l*)))
 ∧ listp (lr-expr-list (*l*))
 ∧ (car (cur-expr (butlast (offset (p-pc (*l*))),
                   program-body (p-current-program (*l*))))
   ≠ `'if`))
→ (lr-p-pc-1 (program-body (p-current-program (*l*)), nx (offset (p-pc (*l*))))
  = (lr-p-pc-1 (program-body (p-current-program (*l*)), offset (p-pc (*l*)))
    + lr-p-c-size (**t**, lr-expr (*l*)))))

EVENT: Disable lr-p-pc-1-listp-offset.

THEOREM: lr-p-pc-1-dv-1-car-lr-expr-if
((car (lr-expr (*l*)) = 'if)
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (*body* = program-body (p-current-program (*l*))))
→ (lr-p-pc-1 (*body*, dv (offset (p-pc (*l*)), 1))
    = lr-p-pc-1 (*body*, offset (p-pc (*l*))))

THEOREM: lr-p-pc-1-dv-2-car-lr-expr-if
(listp (lr-expr (*l*))
 ∧ (car (lr-expr (*l*)) = 'if)
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*))
→ (lr-p-pc-1 (program-body (p-current-program (*l*)), dv (offset (p-pc (*l*)), 2))
    = (3
       + lr-p-c-size (**t**, cadr (lr-expr (*l*)))
       + lr-p-pc-1 (program-body (p-current-program (*l*)),
                    offset (p-pc (*l*))))))

THEOREM: lr-p-pc-1-dv-3-car-lr-expr-if
(listp (lr-expr (*l*))
 ∧ (car (lr-expr (*l*)) = 'if)
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*))
→ (lr-p-pc-1 (program-body (p-current-program (*l*)), dv (offset (p-pc (*l*)), 3))
    = (lr-p-c-size (**t**, cadr (lr-expr (*l*)))
       + 3
       + lr-p-c-size (**t**, caddr (lr-expr (*l*)))
       + 1
       + lr-p-pc-1 (program-body (p-current-program (*l*)),
                    offset (p-pc (*l*))))))

THEOREM: lr-p-c-size-not-1-car-if
(listp (*expr*) ∧ (car (*expr*) = 'if)) → (lr-p-c-size (**t**, *expr*) ≠ 1)

THEOREM: lr-p-c-size-ge-plus-2-size-cadr-car-if
(listp (*expr*) ∧ (car (*expr*) = 'if))
→ ((1 + (1 + lr-p-c-size (**t**, cadr (*expr*)))) < lr-p-c-size (**t**, *expr*))

THEOREM: get-comp-if-helper-helper
((*n* ≮ (*x* + *y* + 3)) ∧ (*n* ≠ (*x* + *y* + 3)))
→ (get (*n* − (*x* + 3 + *y*), cons (*w*, *l3*))
    = get (*n* − (*x* + *y* + 4), *l3*))

THEOREM: get-comp-if-helper

```
get (n, append (l1, append (list (x, y, z), append (l2, cons (w, l3)))))
=   if n < length (l1) then get (n, l1)
    elseif n < (length (l1) + 3)
    then get (n - length (l1), list (x, y, z))
    elseif n < (length (l1) + length (l2) + 3)
    then get (n - (length (l1) + 3), l2)
    elseif n = (length (l1) + length (l2) + 3) then w
    else get (n - (length (l1) + length (l2) + 4), l3) endif
```

THEOREM: get-comp-if

```
get (n, comp-if (test-instrs, then-instrs, else-instrs, m))
=   if n < length (test-instrs) then get (n, test-instrs)
    elseif n < (length (test-instrs) + 3)
    then get (n - length (test-instrs),
              list (identity (list ('push-constant, LR-F-ADDR)),
                    '(eq),
                    list ('test-bool-and-jump,
                          't,
                          lr-make-label (m
                                         + 4
                                         + length (test-instrs)
                                         + length (then-instrs)))))
    elseif n < (length (test-instrs) + length (then-instrs) + 3)
    then get (n - (length (test-instrs) + 3), then-instrs)
    elseif n = (length (test-instrs) + length (then-instrs) + 3)
    then list ('jump,
               lr-make-label (m
                              + 4
                              + length (test-instrs)
                              + length (then-instrs)
                              + length (else-instrs)))
    else get (n - (length (test-instrs)
                   + length (then-instrs)
                   + 4),
              else-instrs) endif
```

DEFINITION:

```
p-final-pc (flag, l, n)
=   if flag = 'list
    then add-addr (lr-p-pc (l), n + lr-p-c-size ('list, lr-expr-list (l)))
    else add-addr (lr-p-pc (l), n + lr-p-c-size (flag, lr-expr (l))) endif
```

EVENT: Disable p-final-pc.

THEOREM: proper-p-statep-p-set-pc

```
(proper-p-statep (p) ∧ (area-name (p-pc (p)) = area-name (pc)))
→   (proper-p-statep (p-set-pc (p, pc)) = p-objectp-type ('pc, pc, p))

THEOREM: proper-p-statep-p-set-pc-equal-p-set-pc
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧   lr-programs-properp (l, table)
 ∧   (p (lr->p (l), p-clock1 (t, l, c))
      =   p-set-pc (lr->p (lr-eval (t, l, c)),
                    tag ('pc,
                         cons (area-name (p-pc (l)),
                               lr-p-pc-1 (program-body (p-current-program (l)),
                                          offset (p-pc (l)))
                               +   lr-p-c-size (t, lr-expr (l))))))
 ∧   proper-p-statep (lr->p (lr-eval (t, l, c))))
→   proper-p-statep (p (lr->p (l), p-clock1 (t, l, c)))

THEOREM: lr-eval-p-pc-equivalence-helper-1
p (lr->p (l1), p-clock1 (flag1, l2, c1) + p-clock1 (flag2, l3, c2))
=   p (p (lr->p (l1), p-clock1 (flag1, l2, c1)), p-clock1 (flag2, l3, c2))

THEOREM: lr-eval-p-pc-equivalence-helper-1-5
(listp (offset (p-pc (l))) ∧ listp (lr-expr-list (l)))
→   (p-final-pc ('list,
                 lr-set-expr (lr-eval (t, l, c), l, nx (offset (p-pc (l)))),
                 0)
      =   tag ('pc,
               cons (area-name (p-pc (l)),
                     lr-p-pc-1 (program-body (p-current-program (l)),
                                nx (offset (p-pc (l))))
                     +   lr-p-c-size-list (length (lr-expr-list (l)) − 1,
                                           lr-expr-list (l)))))

THEOREM: lr-eval-p-pc-equivalence-helper-2
((p-psw (l) = 'run)
 ∧   listp (offset (p-pc (l)))
 ∧   (¬ listp (lr-expr-list (l))))
→   (p-set-pc (lr->p (l),
                tag ('pc,
                     cons (area-name (p-pc (l)),
                           lr-p-pc-1 (program-body (p-current-program (l)),
                                      offset (p-pc (l)))))))
      =   p (lr->p (l), 0))

THEOREM: lr-eval-p-pc-equivalence-helper-3
((p-psw (l) = 'run)
```

$\wedge$ (*flag* $\neq$ 'list)
$\wedge$ litatom (lr-expr (*l*))
$\wedge$ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ (length (p-temp-stk (*l*)) < p-max-temp-stk-size (*l*)))
$\rightarrow$ (p-set-pc (lr->p (lr-set-tstk (*l*,
cons (cdr (assoc (lr-expr (*l*),
bindings (car (p-ctrl-stk (*l*))))),
p-temp-stk (*l*)))),
p-final-pc (*flag*, *l*, 0))
= p (lr->p (*l*), 1))

THEOREM: lr->p-lr-set-pos-dv-1-car-lr-expr-if
((car (lr-expr (*l*)) = 'if)
$\wedge$ listp (lr-expr (*l*))
$\wedge$ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*))
$\rightarrow$ (lr->p (lr-set-pos (*l*, dv (offset (p-pc (*l*)), 1))) = lr->p (*l*))

THEOREM: lr-p-proper-statep-listp-p-temp-stk-type-car-addr
(lr-p-proper-statep (*temp-stk*, *ctrl-stk*, *data-seg*, *table*) $\wedge$ listp (*temp-stk*))
$\rightarrow$ (type (car (*temp-stk*)) = 'addr)

THEOREM: proper-p-statep-lessp-length-p-temp-stk-max-temp-stk-size
proper-p-statep (lr->p (*l*))
$\rightarrow$ (p-max-temp-stk-size (*l*) $\not<$ length (p-temp-stk (*l*)))

THEOREM: length-p-temp-stk-lr-eval-lr-set-pos
(proper-p-statep (lr->p (*l*))
$\wedge$ good-posp1 (*pos*, program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ lr-proper-formalsp (cdr (p-prog-segment (*l*)))
$\wedge$ (p-psw (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)) = 'run))
$\rightarrow$ (length (p-temp-stk (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)))
= (1 + length (p-temp-stk (*l*))))

THEOREM: not-lessp-length-proper-p-statep-lr-eval-lr-set-pos
(good-posp1 (*pos*, program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ proper-p-statep (lr->p (*l*))
$\wedge$ (length (p-temp-stk (*l*)) $\not<$ (p-max-temp-stk-size (*l*) $-$ 1))
$\wedge$ lr-proper-formalsp (cdr (p-prog-segment (*l*))))
$\rightarrow$ (p-psw (lr-if-ok (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*))) $\neq$ 'run)

THEOREM: lr-pop-tstk-lr-if-ok
(p-psw (lr-if-ok (*l*)) = 'run)
$\rightarrow$ (lr-pop-tstk (lr-if-ok (*l*)) = lr-pop-tstk (*l*))

THEOREM: add-addr-p-final-pc
(add-addr (p-final-pc (*flag*, *l*, *n*), 1 + *m*)
 = add-addr (p-final-pc (*flag*, *l*, 1 + *n*), *m*))
∧ (add-addr (p-final-pc (*flag*, *l*, *n*), 0) = p-final-pc (*flag*, *l*, *n*))

THEOREM: lessp-3-lr-p-c-size-car-if
(listp (*expr*) ∧ (car (*expr*) = 'if)) → (3 < lr-p-c-size (**t**, *expr*))

THEOREM: comp-body-1-car-expr-if
((car (*expr*) = 'if) ∧ listp (*expr*))
→ (comp-body-1 (**t**, *expr*, *n*)
 = comp-if (comp-body-1 (**t**, cadr (*expr*), *n*),
 comp-body-1 (**t**,
 caddr (*expr*),
 n + 3 + lr-p-c-size (**t**, cadr (*expr*))),
 comp-body-1 (**t**,
 cadddr (*expr*),
 n
 + 4
 + lr-p-c-size (**t**, cadr (*expr*))
 + lr-p-c-size (**t**, caddr (*expr*))),
 n))

THEOREM: get-lr-p-c-size-lessp-restn-lr-p-pc-1-comp-body-1
(good-posp1 (*pos*, program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ listp (lr-expr (*l*))
 ∧ (car (lr-expr (*l*)) = 'if)
 ∧ (*m* < 3)
 ∧ (*name* = area-name (p-pc (*l*)))
 ∧ (*pos* = offset (p-pc (*l*))))
→ (unlabel (get (offset (p-final-pc (**t**, lr-set-pos (*l*, dv (*pos*, 1)), *m*)),
 program-body (assoc (*name*, comp-programs (p-prog-segment (*l*))))))
 = get (*m*,
 list (list ('push-constant, LR-F-ADDR),
 '(eq),
 list ('test-bool-and-jump,
 't,
 lr-make-label (lr-p-pc-1 (program-body (p-current-program (*l*)),
 pos)
 + 4
 + lr-p-c-size (**t**,
 cadr (lr-expr (*l*)))
 + lr-p-c-size (**t**,
 caddr (lr-expr (*l*))))))))

THEOREM: area-name-p-final-pc
area-name (p-final-pc (*flag*, *l*, *n*)) = area-name (p-pc (*l*))

THEOREM: lr-eval-p-pc-equivalence-helper-4-helper-1
let *test* **be** lr-eval (**t**, lr-set-pos (*l*, dv (offset (p-pc (*l*)), 1)), *c*)
in
((p-psw (*l*) = 'run)
 ∧ (*c* ≠ 0)
 ∧ listp (lr-expr (*l*))
 ∧ (car (lr-expr (*l*)) = 'if)
 ∧ (p-psw (lr-if-ok (*test*)) = 'run)
 ∧ (car (p-temp-stk (*test*)) ≠ LR-F-ADDR)
 ∧ proper-p-statep (lr->p (*l*))
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ lr-p-proper-statep (p-temp-stk (*l*),
 p-ctrl-stk (*l*),
 p-data-segment (*l*),
 table)
 ∧ lr-proper-formalsp (cdr (p-prog-segment (*l*)))
 ∧ (p-psw (lr-eval (**t**,
 lr-set-expr (lr-pop-tstk (*test*),
 l,
 dv (offset (p-pc (*l*)), 2)),
 c))
 = 'run))
→ (p (p-set-pc (lr->p (*test*),
 p-final-pc (**t**,
 lr-set-pos (*l*, dv (offset (p-pc (*l*)), 1)),
 0)),
 3)
 = p-set-pc (lr->p (lr-pop-tstk (*test*)),
 tag ('pc,
 cons (area-name (p-pc (*l*)),
 3
 + lr-p-c-size (**t**,
 cadr (lr-expr (*l*)))
 + lr-p-pc-1 (program-body (p-current-program (*l*)),
 offset (p-pc (*l*)))))))) **endlet**

THEOREM: lessp-plus-lr-p-c-size-cadr-caddr-3-car-if
(listp (*expr*) ∧ (car (*expr*) = 'if))
→ ((lr-p-c-size (**t**, cadr (*expr*)) + lr-p-c-size (**t**, caddr (*expr*)))
 < (((lr-p-c-size (**t**, *expr*) − 1) − 1) − 1))

THEOREM: get-plus-lr-p-c-size-cadr-caddr-4-comp-body-cur-expr
((*size1* = length (*test*)) ∧ (*size2* = length (*then*)))
→ (get (3 + *size1* + *size2*, comp-if (*test*, *then*, *else*, *n*))
= list ('jump,
lr-make-label (4
+ *n*
+ length (*test*)
+ length (*then*)
+ length (*else*))))

THEOREM: cur-expr-add1-opener
cur-expr (cons (1 + *n*, *pos*), *body*) = cur-expr (*pos*, get (*n*, cdr (*body*)))

THEOREM: lr-p-pc-1-car-expr-if-2
(listp (*expr*) ∧ (car (*expr*) = 'if))
→ (lr-p-pc-1 (*expr*, '(2)) = (3 + lr-p-c-size (**t**, cadr (*expr*))))

THEOREM: get-plus-lr-p-pc-1-lr-pc-size-cadr-assoc-comp-body-if-4
(good-posp1 (offset (p-pc (*l1*)), program-body (p-current-program (*l1*)))
∧ lr-programs-properp (*l1*, *table*)
∧ listp (lr-expr (*l1*))
∧ (car (lr-expr (*l1*)) = 'if)
∧ (*pos* = dv (offset (p-pc (*l1*)), 2))
∧ (area-name (p-pc (*l2*)) = area-name (p-pc (*l1*)))
∧ (p-prog-segment (*l2*) = p-prog-segment (*l1*)))
→ (get (offset (p-final-pc (**t**, lr-set-expr (*l2*, *l1*, *pos*), 0)),
program-body (assoc (area-name (p-pc (*l1*)),
comp-programs (p-prog-segment (*l1*)))))
= list ('dl,
lr-make-label (offset (p-final-pc (**t**,
lr-set-expr (*l2*, *l1*, *pos*),
0))),
nil,
list ('jump,
lr-make-label (4
+ lr-p-pc-1 (program-body (p-current-program (*l1*)),
offset (p-pc (*l1*)))
+ lr-p-c-size (**t**,
cadr (lr-expr (*l1*)))
+ lr-p-c-size (**t**,
caddr (lr-expr (*l1*)))
+ lr-p-c-size (**t**,
cadddr (lr-expr (*l1*)))))))

THEOREM: find-label-lr-make-label-label-instrs

$((m \not< n) \wedge (m \in \mathbf{N}) \wedge (m < (n + \text{length}(\mathit{instrs}))))$
$\rightarrow \quad (\text{find-label}(\text{lr-make-label}(m), \text{label-instrs}(\mathit{instrs}, n)) = (m - n))$

THEOREM: find-label-past-else-lr-expr-car-if
$(\text{listp}(\text{lr-expr}(l))$
$\wedge \quad (\text{car}(\text{lr-expr}(l)) = \texttt{'if})$
$\wedge \quad (\mathit{expr} = \text{lr-expr}(l))$
$\wedge \quad (\mathit{body} = \text{program-body}(\text{p-current-program}(l)))$
$\wedge \quad (\mathit{pos} = \text{offset}(\text{p-pc}(l)))$
$\wedge \quad \text{lr-programs-properp}(l, \mathit{table})$
$\wedge \quad \text{good-posp1}(\mathit{pos}, \mathit{body}))$
$\rightarrow \quad (\text{find-label}(\text{lr-make-label}(1 + (1 + (1 + (1 + (\text{lr-p-c-size}(\mathbf{t}, \text{cadr}(\mathit{expr})) + \text{lr-p-c-size}(\mathbf{t}, \text{caddr}(\mathit{expr})) + \text{lr-p-c-size}(\mathbf{t}, \text{cadddr}(\mathit{expr})) + \text{lr-p-pc-1}(\mathit{body}, \mathit{pos}))))))),$
$\text{program-body}(\text{assoc}(\text{area-name}(\text{p-pc}(l)), \text{comp-programs}(\text{p-prog-segment}(l)))))$
$= \quad (1 + (1 + (1 + (1 + (\text{lr-p-c-size}(\mathbf{t}, \text{cadr}(\mathit{expr})) + \text{lr-p-c-size}(\mathbf{t}, \text{caddr}(\mathit{expr})) + \text{lr-p-c-size}(\mathbf{t}, \text{cadddr}(\mathit{expr})) + \text{lr-p-pc-1}(\mathit{body}, \mathit{pos})))))))$

THEOREM: lr-eval-p-pc-equivalence-helper-4-helper-2
let *test* **be** $\text{lr-eval}(\mathbf{t}, \text{lr-set-pos}(l, \text{dv}(\text{offset}(\text{p-pc}(l)), \texttt{1})), c)$
in
let *then* **be** $\text{lr-set-expr}(\text{lr-pop-tstk}(\mathit{test}), l, \text{dv}(\text{offset}(\text{p-pc}(l)), \texttt{2}))$
in
$((\text{p-psw}(l) = \texttt{'run})$
$\wedge \quad (\mathit{flag} \neq \texttt{'list})$
$\wedge \quad (c \neq \texttt{0})$
$\wedge \quad (c \in \mathbf{N})$
$\wedge \quad \text{listp}(\text{lr-expr}(l))$
$\wedge \quad (\text{car}(\text{lr-expr}(l)) = \texttt{'if})$
$\wedge \quad (\text{p-psw}(\text{lr-if-ok}(\mathit{test})) = \texttt{'run})$
$\wedge \quad (\text{car}(\text{p-temp-stk}(\mathit{test})) \neq \text{LR-F-ADDR})$
$\wedge \quad \text{proper-p-statep}(\text{lr->p}(l))$
$\wedge \quad \text{good-posp1}(\text{offset}(\text{p-pc}(l)), \text{program-body}(\text{p-current-program}(l)))$
$\wedge \quad \text{lr-programs-properp}(l, \mathit{table})$
$\wedge \quad \text{lr-p-proper-statep}(\text{p-temp-stk}(l),$

p-ctrl-stk (*l*),
p-data-segment (*l*),
table)
∧ lr-proper-formalsp (cdr (p-prog-segment (*l*)))
∧ (p-psw (lr-eval (**t**, *then*, *c*)) = 'run)
→ (p (p-set-pc (lr->p (lr-eval (**t**, *then*, *c*)),
p-final-pc (**t**, *then*, 0)),
1)
= p-set-pc (lr->p (lr-eval (**t**, *then*, *c*)),
p-final-pc (*flag*, *l*, 0))) **endlet endlet**

THEOREM: lr-eval-p-pc-equivalence-helper-4
let *test* **be** lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*),
cadr-size **be** lr-p-c-size (**t**, cadr (lr-expr (*l*))),
lr-p-pc-1 **be** lr-p-pc-1 (program-body (p-current-program (*l*)),
offset (p-pc (*l*))),
then **be** lr-set-expr (lr-pop-tstk (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)),
l,
dv (offset (p-pc (*l*)), 2))
in
((p-psw (*l*) = 'run)
∧ (*flag* ≠ 'list)
∧ (*c* ≠ 0)
∧ (*c* ∈ **N**)
∧ listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) = 'if)
∧ (p-psw (lr-if-ok (*test*)) = 'run)
∧ (car (p-temp-stk (*test*)) ≠ LR-F-ADDR)
∧ (p (lr->p (*l*), p-clock1 (**t**, lr-set-pos (*l*, *pos*), *c*))
= p-set-pc (lr->p (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)),
p-final-pc (**t**, lr-set-pos (*l*, *pos*), 0)))
∧ (p (p-set-pc (lr->p (lr-pop-tstk (*test*)),
tag ('pc,
cons (area-name (p-pc (*l*)),
3 + *cadr-size* + *lr-p-pc-1*))),
p-clock1 (**t**, *then*, *c*))
= p-set-pc (lr->p (lr-eval (**t**, *then*, *c*)),
p-final-pc (**t**, *then*, 0)))
∧ proper-p-statep (lr->p (*l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ lr-p-proper-statep (p-temp-stk (*l*),
p-ctrl-stk (*l*),
p-data-segment (*l*),

table)
∧ lr-proper-formalsp (cdr (p-prog-segment (*l*)))
∧ (p-psw (lr-eval (**t**, *then*, *c*)) = 'run)
∧ (*pos* = dv (offset (p-pc (*l*)), 1)))
→ (p (lr->p (*l*),
p-clock1 (**t**, lr-set-pos (*l*, *pos*), *c*)
\+ 3
\+ p-clock1 (**t**, *then*, *c*)
\+ 1)
= p-set-pc (lr->p (lr-eval (**t**, *then*, *c*)),
p-final-pc (*flag*, *l*, 0))) **endlet**

THEOREM: lessp-plus-lr-p-pc-1-lr-p-c-size-3-1-lr-expr-car-if
((car (lr-expr (*l*)) = 'if)
∧ listp (lr-expr (*l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*))
→ ((lr-p-c-size (**t**, cadr (lr-expr (*l*)))
\+ 3
\+ lr-p-c-size (**t**, caddr (lr-expr (*l*)))
\+ 1
\+ lr-p-pc-1 (program-body (p-current-program (*l*)), offset (p-pc (*l*))))
< length (program-body (assoc (area-name (p-pc (*l*)),
comp-programs (p-prog-segment (*l*))))))

THEOREM: find-label-else-start-lr-expr-car-if
(listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) = 'if)
∧ (*expr* = lr-expr (*l*))
∧ (*body* = program-body (p-current-program (*l*)))
∧ (*pos* = offset (p-pc (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ good-posp1 (*pos*, *body*))
→ (find-label (lr-make-label (1 + (1 + (1 + (1 + (lr-p-c-size (**t**,
cadr (*expr*))
\+ lr-p-c-size (**t**,
caddr (*expr*))
\+ lr-p-pc-1 (*body*,
pos))))))),
program-body (assoc (area-name (p-pc (*l*)),
comp-programs (p-prog-segment (*l*)))))
= (1 + (1 + (1 + (1 + (lr-p-c-size (**t**, cadr (*expr*))
\+ lr-p-c-size (**t**, caddr (*expr*))
\+ lr-p-pc-1 (*body*, *pos*)))))))

THEOREM: lr-eval-p-pc-equivalence-helper-5-helper-1

```
let test be lr-eval(t, lr-set-pos(l, dv(offset(p-pc(l)), 1)), c)
in
((p-psw(l) = 'run)
 ∧  (c ≠ 0)
 ∧  listp(lr-expr(l))
 ∧  (car(lr-expr(l)) = 'if)
 ∧  (p-psw(lr-if-ok(test)) = 'run)
 ∧  (car(p-temp-stk(test)) = LR-F-ADDR)
 ∧  proper-p-statep(lr->p(l))
 ∧  good-posp1(offset(p-pc(l)), program-body(p-current-program(l)))
 ∧  lr-programs-properp(l, table)
 ∧  lr-p-proper-statep(p-temp-stk(l),
                       p-ctrl-stk(l),
                       p-data-segment(l),
                       table)
 ∧  lr-proper-formalsp(cdr(p-prog-segment(l))))
→  (p(p-set-pc(lr->p(test),
                p-final-pc(t,
                           lr-set-pos(l, dv(offset(p-pc(l)), 1)),
                           0)),
      3)
   =  p-set-pc(lr->p(lr-pop-tstk(test)),
               tag('pc,
                   cons(area-name(p-pc(l)),
                        lr-p-c-size(t, cadr(lr-expr(l)))
                        +  3
                        +  lr-p-c-size(t,
                                       caddr(lr-expr(l)))
                        +  1
                        +  lr-p-pc-1(program-body(p-current-program(l)),
                                     offset(p-pc(l))))))) endlet
```

THEOREM: lr-eval-p-pc-equivalence-helper-5-helper-2

```
let test be lr-eval(t, lr-set-pos(l, dv(offset(p-pc(l)), 1)), c)
in
let else be lr-set-expr(lr-pop-tstk(test), l, dv(offset(p-pc(l)), 3))
in
((flag ≠ 'list)
 ∧  listp(lr-expr(l))
 ∧  (car(lr-expr(l)) = 'if)
 ∧  good-posp1(offset(p-pc(l)),
               program-body(p-current-program(l)))
 ∧  lr-programs-properp(l, table))
```

$\rightarrow$ (p-final-pc (*flag*, *l*, 0) = p-final-pc (**t**, *else*, 0)) **endlet endlet**

THEOREM: lr-eval-p-pc-equivalence-helper-5
let *test* **be** lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*),
cadr-size **be** lr-p-c-size (**t**, cadr (lr-expr (*l*))),
caddr-size **be** lr-p-c-size (**t**, caddr (lr-expr (*l*))),
lr-p-pc-1 **be** lr-p-pc-1 (program-body (p-current-program (*l*)),
offset (p-pc (*l*))),
else **be** lr-set-expr (lr-pop-tstk (lr-eval (**t**, lr-set-pos (*l*, *pos*), *c*)),
l,
dv (offset (p-pc (*l*)), 3))
in
((p-psw (*l*) = 'run)
$\wedge$ (*flag* $\neq$ 'list)
$\wedge$ (*c* $\neq$ 0)
$\wedge$ (*c* $\in$ **N**)
$\wedge$ listp (lr-expr (*l*))
$\wedge$ (car (lr-expr (*l*)) = 'if)
$\wedge$ (p-psw (lr-if-ok (*test*)) = 'run)
$\wedge$ (car (p-temp-stk (*test*)) = LR-F-ADDR)
$\wedge$ (p (p-set-pc (lr->p (lr-pop-tstk (*test*)),
tag ('pc,
cons (area-name (p-pc (*l*)),
cadr-size
+ 3
+ *caddr-size*
+ 1
+ *lr-p-pc-1*))),
p-clock1 (**t**, *else*, *c*))
= p-set-pc (lr->p (lr-eval (**t**, *else*, *c*)),
p-final-pc (**t**, *else*, 0)))
$\wedge$ (p (lr->p (*l*), p-clock1 (**t**, lr-set-pos (*l*, *pos*), *c*))
= p-set-pc (lr->p (*test*),
p-final-pc (**t**, lr-set-pos (*l*, *pos*), 0)))
$\wedge$ proper-p-statep (lr->p (*l*))
$\wedge$ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
$\wedge$ lr-programs-properp (*l*, *table*)
$\wedge$ lr-p-proper-statep (p-temp-stk (*l*),
p-ctrl-stk (*l*),
p-data-segment (*l*),
table)
$\wedge$ lr-proper-formalsp (cdr (p-prog-segment (*l*)))
$\wedge$ (p-psw (lr-eval (**t**, *else*, *c*)) = 'run)
$\wedge$ (*pos* = dv (offset (p-pc (*l*)), 1)))

```
→   (p (lr->p (l),
        p-clock1 (t, lr-set-pos (l, pos), c)
      +  3
      +  p-clock1 (t, else, c))
    =  p-set-pc (lr->p (lr-eval (t, else, c)),
                 p-final-pc (flag, l, 0))) endlet
```

THEOREM: lr-p-pc-1-dv-1-car-lr-expr-temp-eval

```
((car (lr-expr (l)) = S-TEMP-EVAL)
 ∧  listp (lr-expr (l))
 ∧  good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  lr-programs-properp (l, table)
 ∧  (body = program-body (p-current-program (l))))
→   (lr-p-pc-1 (body, dv (offset (p-pc (l)), 1))
    =  lr-p-pc-1 (body, offset (p-pc (l))))
```

THEOREM: comp-body-1-car-expr-temp-eval

```
(listp (expr) ∧ (car (expr) = S-TEMP-EVAL))
→   (comp-body-1 (t, expr, n)
    =  append (comp-body-1 (t, cadr (expr), n),
              list (list ('set-local, caddr (expr)))))
```

THEOREM: lr-eval-p-pc-equivalence-helper-5-get-lr-p-c-size

```
(listp (lr-expr (l))
 ∧  (car (lr-expr (l)) = S-TEMP-EVAL)
 ∧  good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  lr-programs-properp (l, table)
 ∧  (body = program-body (p-current-program (l)))
 ∧  (progs = p-prog-segment (l)))
→   (unlabel (get (lr-p-pc-1 (body, offset (p-pc (l)))
                 +  lr-p-c-size (t, cadr (lr-expr (l))),
                 program-body (assoc (area-name (p-pc (l)),
                                      comp-programs (progs)))))
    =  list ('set-local, caddr (lr-expr (l))))
```

THEOREM: lr->p-lr-set-pos-dv-1-car-lr-expr-temp-eval

```
(listp (lr-expr (l))
 ∧  (car (lr-expr (l)) = S-TEMP-EVAL)
 ∧  good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  lr-programs-properp (l, table))
→   (lr->p (lr-set-pos (l, dv (offset (p-pc (l)), 1))) = lr->p (l))
```

THEOREM: lr-eval-p-pc-equivalence-helper-6-helper

```
let lr-eval be lr-eval (t, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
in
```

```
((p-psw (l) = 'run)
 ∧   (flag ≠ 'list)
 ∧   (c ≠ 0)
 ∧   (c ∈ N)
 ∧   listp (lr-expr (l))
 ∧   (car (lr-expr (l)) = S-TEMP-EVAL)
 ∧   proper-p-statep (lr->p (l))
 ∧   good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧   lr-programs-properp (l, table)
 ∧   (p-psw (lr-eval) = 'run))
→   (p (p-set-pc (lr->p (lr-eval),
                  p-final-pc (t,
                              lr-set-pos (l, dv (offset (p-pc (l)), 1)),
                              0)),
       1)
     =   p-set-pc (lr->p (lr-set-temp (lr-eval,
                                       car (p-temp-stk (lr-eval)),
                                       caddr (lr-expr (l)))),
                  p-final-pc (flag, l, 0))) endlet
```

THEOREM: lr-eval-p-pc-equivalence-helper-6

```
((p-psw (l) = 'run)
 ∧   (flag ≠ 'list)
 ∧   (c ≠ 0)
 ∧   (c ∈ N)
 ∧   listp (lr-expr (l))
 ∧   (car (lr-expr (l)) = S-TEMP-EVAL)
 ∧   (p (lr->p (l), p-clock1 (t, lr-set-pos (l, pos), c))
      =   p-set-pc (lr->p (lr-eval (t, lr-set-pos (l, pos), c)),
                    p-final-pc (t, lr-set-pos (l, pos), 0)))
 ∧   proper-p-statep (lr->p (l))
 ∧   good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧   lr-programs-properp (l, table)
 ∧   (p-psw (lr-eval (t, lr-set-pos (l, pos), c)) = 'run)
 ∧   (pos = dv (offset (p-pc (l)), 1)))
→   (p-set-pc (lr->p (lr-set-temp (lr-eval (t, lr-set-pos (l, pos), c),
                                    car (p-temp-stk (lr-eval (t,
                                                              lr-set-pos (l, pos),
                                                              c))),
                                    caddr (lr-expr (l)))),
                  p-final-pc (flag, l, 0))
     =   p (lr->p (l), p-clock1 (t, lr-set-pos (l, pos), c) + 1))
```

THEOREM: get-comp-temp-test

```
(listp (expr) ∧ (car (expr) = S-TEMP-TEST))
→  (get (m, comp-body-1 (t, expr, n))
    =  if m < 4
       then get (m,
                 list (list ('push-local, caddr (expr)),
                       list ('push-constant,
                             identity (LR-UNDEF-ADDR)),
                       '(eq),
                       list ('test-bool-and-jump,
                             'f,
                             lr-make-label (n
                                            +  6
                                            +  lr-p-c-size (t,
                                                            cadr (expr))))))
       elseif m < (lr-p-c-size (t, cadr (expr)) + 4)
       then get (m − 4, comp-body-1 (t, cadr (expr), n + 4))
       else get (m − (lr-p-c-size (t, cadr (expr)) + 4),
                 list (list ('set-local, caddr (expr)),
                       list ('jump,
                             lr-make-label (n
                                            +  7
                                            +  lr-p-c-size (t,
                                                            cadr (expr)))),
                       list ('push-local, caddr (expr)))) endif)

THEOREM: lr-p-c-size-temp-test-opener
(listp (expr) ∧ (car (expr) = S-TEMP-TEST))
→  (lr-p-c-size (t, expr) = (lr-p-c-size (t, cadr (expr)) + 7))

THEOREM: get-+-lr-p-pc-1-lessp-3-temp-test-assoc-comp-programs
(listp (lr-expr (l))
 ∧  (car (lr-expr (l)) = S-TEMP-TEST)
 ∧  good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  lr-programs-properp (l, table)
 ∧  (n < 4)
 ∧  (progs = p-prog-segment (l))
 ∧  (body = program-body (p-current-program (l)))
 ∧  (pos = offset (p-pc (l))))
→  (unlabel (get (lr-p-pc-1 (body, pos) + n,
                  program-body (assoc (area-name (p-pc (l)),
                                       comp-programs (progs))))))
    =  get (n,
            list (list ('push-local, caddr (lr-expr (l))),
                  identity (list ('push-constant, LR-UNDEF-ADDR)),
```

```
        '(eq),
        list ('test-bool-and-jump,
              'f,
              lr-make-label (lr-p-pc-1 (body, pos)
                                       +  6
                                       +  lr-p-c-size (t,
                                                       cadr (lr-expr (l))))))))
```

THEOREM: car-comp-body-lr-expr-3-temp-test
(listp (*expr*) ∧ (car (*expr*) = S-TEMP-TEST))
→ (car (comp-body-1 (**t**, *expr*, *n*)) = list ('push-local, caddr (*expr*)))

THEOREM: get-+-lr-p-pc-1-n-2-size-temp-test-assoc-comp-programs
(listp (lr-expr (*l*))
∧ (car (lr-expr (*l*)) = S-TEMP-TEST)
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ (*n* ≮ 4)
∧ (6 ≮ *n*)
∧ (*progs* = p-prog-segment (*l*))
∧ (*body* = program-body (p-current-program (*l*)))
∧ (*pos* = offset (p-pc (*l*))))
→ (unlabel (get (lr-p-pc-1 (*body*, *pos*)
+ *n*
+ lr-p-c-size (**t**, cadr (lr-expr (*l*))),
program-body (assoc (area-name (p-pc (*l*)),
comp-programs (*progs*)))))
= get (*n* − 4,
list (list ('set-local, caddr (lr-expr (*l*))),
list ('jump,
lr-make-label (lr-p-pc-1 (*body*, *pos*)
+ 7
+ lr-p-c-size (**t**,
cadr (lr-expr (*l*)))))),
list ('push-local, caddr (lr-expr (*l*))))))

EVENT: Disable get-comp-temp-test.

EVENT: Disable lr-p-c-size-temp-test-opener.

THEOREM: definedp-caddr-lr-expr-bindings-ctrl-stk-rewrite
(lr-programs-properp (*l*, *table*)
∧ ((car (lr-expr (*l*)) = S-TEMP-FETCH)
∨ (car (lr-expr (*l*)) = S-TEMP-TEST))

$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ proper-p-statep (lr->p (l)))
$\rightarrow$ definedp (caddr (lr-expr (l)), bindings (car (p-ctrl-stk (l))))

THEOREM: member-lr-good-pointerps-type-addr-untag-whole
(($addr \in list$)
$\wedge$ (type ($addr$) = `'addr`)
$\wedge$ (untag ($addr$) = $rest$)
$\wedge$ ($addr \neq$ list (`'addr`, $rest$)))
$\rightarrow$ ($\neg$ lr-good-pointerps ($list$, $data\text{-}seg$))

THEOREM: find-label-temp-test-end-lr-expr-car-temp-test
(listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) = S-TEMP-TEST)
$\wedge$ ($body$ = program-body (p-current-program (l)))
$\wedge$ (pos = offset (p-pc (l)))
$\wedge$ ($expr$ = lr-expr (l))
$\wedge$ lr-programs-properp (l, $table$)
$\wedge$ good-posp1 (pos, $body$)
$\wedge$ (`7` $\not<$ n))
$\rightarrow$ (find-label (lr-make-label (lr-p-pc-1 ($body$, pos)
+ n
+ lr-p-c-size (**t**, cadr ($expr$))),
program-body (assoc (area-name (p-pc (l)),
comp-programs (p-prog-segment (l))))))
= (lr-p-pc-1 ($body$, pos) + n + lr-p-c-size (**t**, cadr ($expr$))))

THEOREM: lr-p-proper-statep-lr-good-pointerps-strip-cdrs-binding
(lr-p-proper-statep (p-temp-stk (l), p-ctrl-stk (l), p-data-segment (l), $table$)
$\wedge$ proper-p-statep (lr->p (l)))
$\rightarrow$ lr-good-pointerps (strip-cdrs (bindings (car (p-ctrl-stk (l)))),
p-data-segment (l))

THEOREM: lr-eval-p-pc-equivalence-helper-7
(($flag \neq$ `'list`)
$\wedge$ listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) = S-TEMP-TEST)
$\wedge$ (p-max-temp-stk-size (l) $\not<$ (`2` + length (p-temp-stk (l))))
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, $table$)
$\wedge$ lr-p-proper-statep (p-temp-stk (l),
p-ctrl-stk (l),
p-data-segment (l),
$table$)

$\wedge$ (p-psw (lr-do-temp-fetch (l)) = 'run))
$\rightarrow$ (p-set-pc (lr->p (lr-do-temp-fetch (l)), p-final-pc ($flag$, l, 0))
= p (lr->p (l), 5))

THEOREM: lr-p-pc-dv-1-s-temp-test
(listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) = S-TEMP-TEST)
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, $table$))
$\rightarrow$ (lr-p-pc-1 (program-body (p-current-program (l)), dv (offset (p-pc (l)), 1))
= (lr-p-pc-1 (program-body (p-current-program (l)), offset (p-pc (l)))
+ 4))

THEOREM: lr-eval-p-pc-equivalence-helper-8-helper-1
((p-psw (l) = 'run)
$\wedge$ listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) = S-TEMP-TEST)
$\wedge$ (p-max-temp-stk-size (l) $\not<$ (2 + length (p-temp-stk (l))))
$\wedge$ (local-var-value (caddr (lr-expr (l)), p-ctrl-stk (l)) = LR-UNDEF-ADDR)
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, $table$))
$\rightarrow$ (lr->p (lr-set-pos (l, dv (offset (p-pc (l)), 1))) = p (lr->p (l), 4))

THEOREM: lr-eval-p-pc-equivalence-helper-8-helper-2-helper
(4 + lr-p-c-size (**t**, $expr$) + 1) = (5 + lr-p-c-size (**t**, $expr$))

THEOREM: lr-eval-p-pc-equivalence-helper-8-helper-2
let lr-$eval$ **be** lr-eval (**t**, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
in
(($flag$ $\neq$ 'list)
$\wedge$ listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) = S-TEMP-TEST)
$\wedge$ (p-max-temp-stk-size (l) $\neq$ 0)
$\wedge$ (p-max-temp-stk-size (l) $\in$ **N**)
$\wedge$ (p-max-temp-stk-size (l) $\neq$ 1)
$\wedge$ (((p-max-temp-stk-size (l) $-$ 1) $-$ 1) $\not<$ length (p-temp-stk (l)))
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, $table$)
$\wedge$ (p-psw (lr-$eval$) = 'run))
$\rightarrow$ (p-set-pc (lr->p (lr-set-temp (lr-$eval$,
car (p-temp-stk (lr-$eval$)),
caddr (lr-expr (l)))),
p-final-pc ($flag$, l, 0))

```
    =   p (p-set-pc (lr->p (lr-eval),
                     p-final-pc (t,
                                 lr-set-pos (l,
                                             dv (offset (p-pc (l)), 1)),
                                 0)),
          2)) endlet
```

EVENT: Disable lr-eval-p-pc-equivalence-helper-8-helper-2-helper.

THEOREM: lr-eval-p-pc-equivalence-helper-8

```
((p-psw (l) = 'run)
 ∧  (flag ≠ 'list)
 ∧  listp (lr-expr (l))
 ∧  (car (lr-expr (l)) = S-TEMP-TEST)
 ∧  (p-max-temp-stk-size (l) ≮ (2 + length (p-temp-stk (l))))
 ∧  (¬ lr-eval-temp-setp (l))
 ∧  (p (lr->p (lr-set-pos (l, pos)), p-clock1 (t, lr-set-pos (l, pos), c))
     =  p-set-pc (lr->p (lr-eval (t, lr-set-pos (l, pos), c)),
                 p-final-pc (t, lr-set-pos (l, pos), 0)))
 ∧  proper-p-statep (lr->p (l))
 ∧  good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  lr-programs-properp (l, table)
 ∧  lr-p-proper-statep (p-temp-stk (l),
                        p-ctrl-stk (l),
                        p-data-segment (l),
                        table)
 ∧  (p-psw (lr-eval (t, lr-set-pos (l, pos), c)) = 'run)
 ∧  (pos = dv (offset (p-pc (l)), 1)))
→   (p-set-pc (lr->p (lr-set-temp (lr-eval (t, lr-set-pos (l, pos), c),
                                  car (p-temp-stk (lr-eval (t,
                                                            lr-set-pos (l, pos),
                                                            c))),
                                  caddr (lr-expr (l)))),
                   p-final-pc (flag, l, 0))
    =  p (lr->p (l), 4 + p-clock1 (t, lr-set-pos (l, pos), c) + 2))
```

THEOREM: comp-body-1-car-expr-temp-fetch

```
(listp (expr) ∧ (car (expr) = S-TEMP-FETCH))
→   (comp-body-1 (t, expr, n) = list (list ('push-local, caddr (expr))))
```

THEOREM: get-lr-p-pc-1-comp-body-1-temp-fetch

```
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 ∧  listp (lr-expr (l))
 ∧  (car (lr-expr (l)) = S-TEMP-FETCH)
```

$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (*prog* = assoc (area-name (p-pc (l)), p-prog-segment (l))))
$\rightarrow$ (get (lr-p-pc-1 (program-body (*prog*), offset (p-pc (l))),
comp-body-1 (**t**, program-body (*prog*), `0`))
= list (`'push-local`, caddr (lr-expr (l))))

THEOREM: lr-eval-p-pc-equivalence-helper-9
((p-psw (l) = `'run`)
$\wedge$ (*flag* $\neq$ `'list`)
$\wedge$ listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) = S-TEMP-FETCH)
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-do-temp-fetch (l)) = `'run`))
$\rightarrow$ (p-set-pc (lr->p (lr-do-temp-fetch (l)), p-final-pc (*flag*, l, `0`))
= p (lr->p (l), `1`))

THEOREM: lr-eval-p-pc-equivalence-helper-10
((p-psw (l) = `'run`)
$\wedge$ (*flag* $\neq$ `'list`)
$\wedge$ listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) = `'quote`)
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (length (p-temp-stk (l)) < p-max-temp-stk-size (l)))
$\rightarrow$ (p-set-pc (lr->p (lr-set-tstk (l, cons (cadr (lr-expr (l)), p-temp-stk (l))))),
p-final-pc (*flag*, l, `0`))
= p (lr->p (l), `1`))

THEOREM: lr-expr-cur-expr-if-same
(car (cur-expr (offset (p-pc (l)), program-body (p-current-program (l)))) = `'if`)
= (car (lr-expr (l)) = `'if`)

THEOREM: lr-p-pc-lr-set-pos-dv-1-car-lr-expr-funcall
(listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) $\neq$ `'if`)
$\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-TEST)
$\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-FETCH)
$\wedge$ (car (lr-expr (l)) $\neq$ `'quote`)
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*))
$\rightarrow$ (lr-p-pc (lr-set-pos (l, dv (offset (p-pc (l)), `1`))) = lr-p-pc (l))

THEOREM: lr->p-lr-set-pos-dv-1-car-lr-expr-funcall

(listp (lr-expr (l))
 $\wedge$ (car (lr-expr (l)) $\neq$ 'if)
 $\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-EVAL)
 $\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-TEST)
 $\wedge$ (car (lr-expr (l)) $\neq$ S-TEMP-FETCH)
 $\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
 $\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 $\wedge$ lr-programs-properp (l, *table*))
$\rightarrow$ (lr->p (lr-set-pos (l, dv (offset (p-pc (l)), 1))) = lr->p (l))

THEOREM: p-set-pc-lr->p-equal-p-fact
(p-set-pc (lr->p (p-state (*pc1*,
ctrl-stk,
temp-stk,
prog-seg,
data-seg,
max-ctrl,
max-temp,
word-size,
psw)),
pc2)
= p)
= ((p-pc (p) = *pc2*)
 $\wedge$ (p-ctrl-stk (p) = *ctrl-stk*)
 $\wedge$ (p-temp-stk (p) = *temp-stk*)
 $\wedge$ (p-data-segment (p) = *data-seg*)
 $\wedge$ (p-prog-segment (p) = comp-programs (*prog-seg*))
 $\wedge$ (p-max-ctrl-stk-size (p) = *max-ctrl*)
 $\wedge$ (p-max-temp-stk-size (p) = *max-temp*)
 $\wedge$ (p-word-size (p) = *word-size*)
 $\wedge$ (p-psw (p) = *psw*))

EVENT: Disable p-set-pc-lr->p-equal-p-fact.

THEOREM: lr->p-p-run-subr-p-run-subr-clock
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
 $\wedge$ lr-programs-properp (l, *table*)
 $\wedge$ listp (lr-expr (l))
 $\wedge$ (car (lr-expr (l)) $\neq$ 'if)
 $\wedge$ subrp (car (lr-expr (l))))
$\rightarrow$ (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p (*new-l*),
add-addr (lr-p-pc (l),
lr-p-c-size-list (arity (car (lr-expr (l))),

```
                                                        lr-expr (l)))))
    =   p (p-set-pc (lr->p (new-l),
                    add-addr (lr-p-pc (l),
                              lr-p-c-size-list (arity (car (lr-expr (l))),
                                                lr-expr (l)))),
          p-run-subr-clock (l, new-l)))
```

THEOREM: p-pc-run-car

```
(proper-p-statep (lr->p (l))
 ∧  (p-psw (l) = 'run)
 ∧  (p-psw (p (p-set-pc (lr->p (l), pc), p-car-clock (p-set-pc (lr->p (l), pc))))
     =  'run)
 ∧  lr-programs-properp (l, table)
 ∧  (unlabel (get (offset (pc),
                  program-body (assoc (area-name (pc),
                                       p-prog-segment (lr->p (l))))))
     =  '(call car)))
→   (p-pc (p (p-set-pc (lr->p (l), pc), p-car-clock (p-set-pc (lr->p (l), pc))))
     =  add-addr (pc, 1))
```

THEOREM: p-pc-run-cdr

```
(proper-p-statep (lr->p (l))
 ∧  (p-psw (l) = 'run)
 ∧  (p-psw (p (p-set-pc (lr->p (l), pc), p-cdr-clock (p-set-pc (lr->p (l), pc))))
     =  'run)
 ∧  lr-programs-properp (l, table)
 ∧  (unlabel (get (offset (pc),
                  program-body (assoc (area-name (pc),
                                       p-prog-segment (lr->p (l))))))
     =  '(call cdr)))
→   (p-pc (p (p-set-pc (lr->p (l), pc), p-cdr-clock (p-set-pc (lr->p (l), pc))))
     =  add-addr (pc, 1))
```

THEOREM: p-pc-run-cons

```
(proper-p-statep (lr->p (l))
 ∧  (p-psw (l) = 'run)
 ∧  (p-psw (p (p-set-pc (lr->p (l), pc), p-cons-clock (p-set-pc (lr->p (l), pc))))
     =  'run)
 ∧  lr-programs-properp (l, table)
 ∧  (unlabel (get (offset (pc),
                  program-body (assoc (area-name (pc),
                                       p-prog-segment (lr->p (l))))))
     =  '(call cons)))
→   (p-pc (p (p-set-pc (lr->p (l), pc), p-cons-clock (p-set-pc (lr->p (l), pc))))
     =  add-addr (pc, 1))
```

THEOREM: p-pc-run-false
(proper-p-statep (lr->p (*l*))
 ∧ (p-psw (*l*) = 'run)
 ∧ (p-psw (p (p-set-pc (lr->p (*l*), *pc*), p-false-clock (p-set-pc (lr->p (*l*), *pc*))))
 = 'run)
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (unlabel (get (offset (*pc*),
 program-body (assoc (area-name (*pc*),
 p-prog-segment (lr->p (*l*))))))
 = '(call false)))
→ (p-pc (p (p-set-pc (lr->p (*l*), *pc*), p-false-clock (p-set-pc (lr->p (*l*), *pc*))))
 = add-addr (*pc*, 1))

THEOREM: p-pc-run-falsep
(proper-p-statep (lr->p (*l*))
 ∧ (p-psw (*l*) = 'run)
 ∧ (p-psw (p (p-set-pc (lr->p (*l*), *pc*), p-falsep-clock (p-set-pc (lr->p (*l*), *pc*))))
 = 'run)
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (unlabel (get (offset (*pc*),
 program-body (assoc (area-name (*pc*),
 p-prog-segment (lr->p (*l*))))))
 = '(call falsep)))
→ (p-pc (p (p-set-pc (lr->p (*l*), *pc*), p-falsep-clock (p-set-pc (lr->p (*l*), *pc*))))
 = add-addr (*pc*, 1))

THEOREM: p-pc-run-listp
(proper-p-statep (lr->p (*l*))
 ∧ (p-psw (*l*) = 'run)
 ∧ (p-psw (p (p-set-pc (lr->p (*l*), *pc*), p-listp-clock (p-set-pc (lr->p (*l*), *pc*))))
 = 'run)
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (unlabel (get (offset (*pc*),
 program-body (assoc (area-name (*pc*),
 p-prog-segment (lr->p (*l*))))))
 = '(call listp)))
→ (p-pc (p (p-set-pc (lr->p (*l*), *pc*), p-listp-clock (p-set-pc (lr->p (*l*), *pc*))))
 = add-addr (*pc*, 1))

THEOREM: p-pc-run-nlistp
(proper-p-statep (lr->p (*l*))
 ∧ (p-psw (*l*) = 'run)
 ∧ (p-psw (p (p-set-pc (lr->p (*l*), *pc*), p-nlistp-clock (p-set-pc (lr->p (*l*), *pc*))))
 = 'run)
 ∧ lr-programs-properp (*l*, *table*)

$\wedge$ (unlabel (get (offset (pc),
program-body (assoc (area-name (pc),
p-prog-segment (lr->p (l))))))
= '(call nlistp)))
$\rightarrow$ (p-pc (p (p-set-pc (lr->p (l), pc), p-nlistp-clock (p-set-pc (lr->p (l), pc))))
= add-addr (pc, 1))

THEOREM: p-pc-run-true
(proper-p-statep (lr->p (l))
$\wedge$ (p-psw (l) = 'run)
$\wedge$ (p-psw (p (p-set-pc (lr->p (l), pc), p-true-clock (p-set-pc (lr->p (l), pc))))
= 'run)
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (unlabel (get (offset (pc),
program-body (assoc (area-name (pc),
p-prog-segment (lr->p (l))))))
= '(call true)))
$\rightarrow$ (p-pc (p (p-set-pc (lr->p (l), pc), p-true-clock (p-set-pc (lr->p (l), pc))))
= add-addr (pc, 1))

THEOREM: p-pc-run-truep
(proper-p-statep (lr->p (l))
$\wedge$ (p-psw (l) = 'run)
$\wedge$ (p-psw (p (p-set-pc (lr->p (l), pc), p-truep-clock (p-set-pc (lr->p (l), pc))))
= 'run)
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (unlabel (get (offset (pc),
program-body (assoc (area-name (pc),
p-prog-segment (lr->p (l))))))
= '(call truep)))
$\rightarrow$ (p-pc (p (p-set-pc (lr->p (l), pc), p-truep-clock (p-set-pc (lr->p (l), pc))))
= add-addr (pc, 1))

THEOREM: p-run-subr-p-pc-add-addr-lr-p-pc-lr-p-c-size
let *pos* **be** dv (offset (p-pc (l)), 1)
in
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ listp (lr-expr (l))
$\wedge$ subrp (car (lr-expr (l)))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ (p-psw (lr-eval ('list, lr-set-pos (l, *pos*), c)) = 'run)
$\wedge$ (p-psw (p-run-subr (car (lr-expr (l)),

p-set-pc (lr->p (lr-eval ('list,
lr-set-pos (l, pos),
c)),
lr-return-pc (l))))
= 'run))
→ (p-pc (p-run-subr (car (lr-expr (l)),
p-set-pc (lr->p (lr-eval ('list,
lr-set-pos (l, pos),
c)),
lr-return-pc (l))))
= add-addr (lr-return-pc (l), 1)) **endlet**

THEOREM: lr-eval-p-pc-equivalence-helper-11
((p-psw (l) = 'run)
∧ ($flag$ ≠ 'list)
∧ (c ≠ 0)
∧ (c ∈ **N**)
∧ listp (lr-expr (l))
∧ (car (lr-expr (l)) ≠ 'if)
∧ (car (lr-expr (l)) ≠ S-TEMP-EVAL)
∧ (car (lr-expr (l)) ≠ S-TEMP-TEST)
∧ (car (lr-expr (l)) ≠ S-TEMP-FETCH)
∧ (car (lr-expr (l)) ≠ 'quote)
∧ subrp (car (lr-expr (l)))
∧ (p-psw (lr-eval ('list, lr-set-pos (l, pos), c)) = 'run)
∧ (p (lr->p (l), p-clock1 ('list, lr-set-pos (l, pos), c))
= p-set-pc (lr->p (lr-eval ('list, lr-set-pos (l, pos), c)),
p-final-pc ('list, lr-set-pos (l, pos), 0)))
∧ proper-p-statep (lr->p (l))
∧ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
∧ lr-programs-properp (l, $table$)
∧ (p-psw (lr-apply-subr (l, lr-eval ('list, lr-set-pos (l, pos), c)))
= 'run)
∧ (pos = dv (offset (p-pc (l)), 1)))
→ (p-set-pc (lr->p (lr-apply-subr (l, lr-eval ('list, lr-set-pos (l, pos), c))),
p-final-pc ($flag$, l, 0))
= p (lr->p (l),
p-clock1 ('list, lr-set-pos (l, pos), c)
+ p-run-subr-clock (l,
lr-eval ('list, lr-set-pos (l, pos), c))))

THEOREM: p-set-pc-twice
p-set-pc (p-set-pc (p, $pc1$), $pc2$) = p-set-pc (p, $pc2$)

THEOREM: lr-eval-p-pc-equivalence-helper-12-helper-1

(good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ listp (lr-expr (*l*))
 ∧ (car (lr-expr (*l*)) ≠ 'if)
 ∧ (car (lr-expr (*l*)) ≠ 'quote)
 ∧ (¬ subrp (car (lr-expr (*l*))))
 ∧ litatom (car (lr-expr (*l*)))
 ∧ (p-psw (lr-eval (**t**, lr-funcall (*l*, *new-l*), *c* − 1)) = 'run)
 ∧ (p-psw (*new-l*) = 'run)
 ∧ (p-prog-segment (*l*) = p-prog-segment (*new-l*)))
→ (p (p-set-pc (lr->p (*new-l*),
 add-addr (lr-p-pc (*l*),
 lr-p-c-size-list (arity (car (lr-expr (*l*))),
 lr-expr (*l*)))),
 1)
 = lr->p (lr-funcall (*l*, *new-l*)))

THEOREM: lr-expr-funcall
((p-psw (lr-funcall (*l*, *new-l*)) = 'run)
 ∧ (p-prog-segment (*l*) = p-prog-segment (*new-l*)))
→ (lr-expr (lr-funcall (*l*, *new-l*))
 = program-body (assoc (user-fname (car (lr-expr (*l*))),
 p-prog-segment (*l*))))

THEOREM: unlabel-car-last-comp-body
unlabel (car (last (comp-body (*body*)))) = '(ret)

THEOREM: unlabel-get-last-funcall-body-assoc-comp-programs
(listp (lr-expr (*l*))
 ∧ (¬ subrp (car (lr-expr (*l*))))
 ∧ (car (lr-expr (*l*)) ≠ 'quote)
 ∧ litatom (car (lr-expr (*l*)))
 ∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
 ∧ lr-programs-properp (*l*, *table*)
 ∧ (*prog-seg* = p-prog-segment (*l*)))
→ (unlabel (get (lr-p-c-size (**t**,
 program-body (assoc (user-fname (car (lr-expr (*l*))),
 prog-seg))),
 program-body (assoc (user-fname (car (lr-expr (*l*))),
 comp-programs (*prog-seg*)))))
 = '(ret))

THEOREM: lr-eval-preserves-cdr-p-ctrl-stk-lr-funcall
(listp (lr-expr (*l*))
 ∧ (¬ subrp (car (lr-expr (*l*)))))

$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
$\wedge$ litatom (car (lr-expr (l)))
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval (**t**, lr-funcall (l, *new-l*), $c - 1$)) = 'run)
$\wedge$ (p-psw (*new-l*) = 'run)
$\wedge$ (*new-l* = lr-eval ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)))
$\rightarrow$ (cdr (p-ctrl-stk (lr-eval (**t**, lr-funcall (l, *new-l*), $c - 1$)))
= p-ctrl-stk (*new-l*))

THEOREM: lr-eval-preserves-ret-pc-car-p-ctrl-stk
(proper-p-statep (lr->p (l))
$\wedge$ good-posp (*flag*, offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval (*flag*, l, c)) = 'run))
$\rightarrow$ (ret-pc (car (p-ctrl-stk (lr-eval (*flag*, l, c)))))
= ret-pc (car (p-ctrl-stk (l))))

THEOREM: lr-eval-preserves-ret-pc-car-p-ctrl-stk-lr-funcall
(listp (lr-expr (l))
$\wedge$ ($\neg$ subrp (car (lr-expr (l))))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
$\wedge$ litatom (car (lr-expr (l)))
$\wedge$ proper-p-statep (lr->p (l))
$\wedge$ good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ (p-psw (lr-eval (**t**, lr-funcall (l, *new-l*), $c - 1$)) = 'run)
$\wedge$ (p-psw (*new-l*) = 'run)
$\wedge$ (*new-l* = lr-eval ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)))
$\rightarrow$ (ret-pc (car (p-ctrl-stk (lr-eval (**t**, lr-funcall (l, *new-l*), $c - 1$)))))
= add-addr (lr-return-pc (l), 1))

THEOREM: lr-eval-p-pc-equivalence-helper-12-helper-2
let *new-l* **be** lr-eval ('list, lr-set-pos (l, dv (offset (p-pc (l)), 1)), c)
in
(good-posp1 (offset (p-pc (l)), program-body (p-current-program (l)))
$\wedge$ lr-programs-properp (l, *table*)
$\wedge$ listp (lr-expr (l))
$\wedge$ (car (lr-expr (l)) $\neq$ 'if)
$\wedge$ (car (lr-expr (l)) $\neq$ 'quote)
$\wedge$ ($\neg$ subrp (car (lr-expr (l))))
$\wedge$ proper-p-statep (lr->p (l))

$$
\begin{array}{ll}
\wedge & \text{litatom}\,(\text{car}\,(\text{lr-expr}\,(l))) \\
\wedge & \text{lr-proper-formalsp}\,(\text{cdr}\,(\text{p-prog-segment}\,(l))) \\
\wedge & (\text{p-psw}\,(\text{lr-eval}\,(\mathbf{t},\ \text{lr-funcall}\,(l,\ \textit{new-l}),\ c-1)) = \texttt{'run}) \\
\wedge & (\text{p-psw}\,(\textit{new-l}) = \texttt{'run})) \\
\rightarrow & (\text{p}\,(\text{p-set-pc}\,(\text{lr->p}\,(\text{lr-eval}\,(\mathbf{t},\ \text{lr-funcall}\,(l,\ \textit{new-l}),\ c-1)), \\
 & \qquad\qquad \text{add-addr}\,(\text{lr-p-pc}\,(\text{lr-funcall}\,(l,\ \textit{new-l})), \\
 & \qquad\qquad\qquad\qquad \text{lr-p-c-size}\,(\mathbf{t}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{lr-expr}\,(\text{lr-funcall}\,(l,\ \textit{new-l}))))), \\
 & \quad \texttt{1}) \\
 & = \quad \text{p-set-pc}\,(\text{lr->p}\,(\text{lr-pop-cstk}\,(\text{lr-eval}\,(\mathbf{t}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{lr-funcall}\,(l,\ \textit{new-l}), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c-1))), \\
 & \qquad\qquad\qquad \text{add-addr}\,(\text{lr-p-pc}\,(l), \\
 & \qquad\qquad\qquad\qquad\qquad \text{lr-p-c-size-list}\,(\text{arity}\,(\text{car}\,(\text{lr-expr}\,(l))), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{lr-expr}\,(l)) \\
 & \qquad\qquad\qquad\qquad\qquad + \quad \texttt{1}))) \ \textbf{endlet}
\end{array}
$$

THEOREM: lr-eval-p-pc-equivalence-helper-12

$$
\begin{array}{ll}
\textbf{let}\ \textit{fs}\ \ \textbf{be}\ \ \text{lr-funcall}\,(l,\ \text{lr-eval}\,(\texttt{'list},\ \text{lr-set-pos}\,(l,\ \textit{pos}),\ c)) & \\
\textbf{in} & \\
((\text{p-psw}\,(l) = \texttt{'run}) & \\
\wedge & (\textit{flag} \neq \texttt{'list}) \\
\wedge & (c \neq \texttt{0}) \\
\wedge & (c \in \mathbf{N}) \\
\wedge & \text{listp}\,(\text{lr-expr}\,(l)) \\
\wedge & (\text{car}\,(\text{lr-expr}\,(l)) \neq \texttt{'if}) \\
\wedge & (\text{car}\,(\text{lr-expr}\,(l)) \neq \text{S-TEMP-EVAL}) \\
\wedge & (\text{car}\,(\text{lr-expr}\,(l)) \neq \text{S-TEMP-TEST}) \\
\wedge & (\text{car}\,(\text{lr-expr}\,(l)) \neq \text{S-TEMP-FETCH}) \\
\wedge & (\text{car}\,(\text{lr-expr}\,(l)) \neq \texttt{'quote}) \\
\wedge & (\neg\ \text{subrp}\,(\text{car}\,(\text{lr-expr}\,(l)))) \\
\wedge & \text{litatom}\,(\text{car}\,(\text{lr-expr}\,(l))) \\
\wedge & (\text{p-psw}\,(\text{lr-eval}\,(\texttt{'list},\ \text{lr-set-pos}\,(l,\ \textit{pos}),\ c)) = \texttt{'run}) \\
\wedge & (\text{p}\,(\text{lr->p}\,(\textit{fs}),\ \text{p-clock1}\,(\mathbf{t},\ \textit{fs},\ c-1)) \\
 & \quad = \quad \text{p-set-pc}\,(\text{lr->p}\,(\text{lr-eval}\,(\mathbf{t},\ \textit{fs},\ c-1)), \\
 & \qquad\qquad \text{p-final-pc}\,(\mathbf{t},\ \textit{fs},\ \texttt{0}))) \\
\wedge & (\text{p}\,(\text{lr->p}\,(l),\ \text{p-clock1}\,(\texttt{'list},\ \text{lr-set-pos}\,(l,\ \textit{pos}),\ c)) \\
 & \quad = \quad \text{p-set-pc}\,(\text{lr->p}\,(\text{lr-eval}\,(\texttt{'list},\ \text{lr-set-pos}\,(l,\ \textit{pos}),\ c)), \\
 & \qquad\qquad \text{p-final-pc}\,(\texttt{'list},\ \text{lr-set-pos}\,(l,\ \textit{pos}),\ \texttt{0}))) \\
\wedge & \text{proper-p-statep}\,(\text{lr->p}\,(l)) \\
\wedge & \text{good-posp1}\,(\text{offset}\,(\text{p-pc}\,(l)),\ \text{program-body}\,(\text{p-current-program}\,(l))) \\
\wedge & \text{lr-programs-properp}\,(l,\ \textit{table}) \\
\wedge & \text{lr-proper-formalsp}\,(\text{cdr}\,(\text{p-prog-segment}\,(l))) \\
\wedge & (\text{p-psw}\,(\text{lr-eval}\,(\mathbf{t},\ \textit{fs},\ c-1)) = \texttt{'run})
\end{array}
$$

```
  ∧   (pos = dv (offset (p-pc (l)), 1)))
→   (p-set-pc (lr->p (lr-pop-cstk (lr-eval (t, fs, c − 1))),
                 p-final-pc (flag, l, 0))
      =   p (lr->p (l),
               p-clock1 ('list, lr-set-pos (l, pos), c)
               +   1
               +   p-clock1 (t, fs, c − 1)
               +   1)) endlet
```

THEOREM: lr-eval-p-pc-equivalence

```
(proper-p-statep (lr->p (l))
  ∧   good-posp (flag, offset (p-pc (l)), program-body (p-current-program (l)))
  ∧   ((flag ≠ 'list)
       ∨   (car (cur-expr (butlast (offset (p-pc (l))),
                                   program-body (p-current-program (l))))
           ≠   'if))
  ∧   lr-programs-properp (l, table)
  ∧   lr-p-proper-statep (p-temp-stk (l),
                          p-ctrl-stk (l),
                          p-data-segment (l),
                          table)
  ∧   lr-proper-formalsp (cdr (p-prog-segment (l)))
  ∧   (p-psw (lr-eval (flag, l, c)) = 'run))
→   (p (lr->p (l), p-clock1 (flag, l, c))
      =   p-set-pc (lr->p (lr-eval (flag, l, c)), p-final-pc (flag, l, 0)))
```

EVENT: Disable lr-expr-cur-expr-if-same.

DEFINITION:

```
logic->p-clock (expr,
                alist,
                program-names,
                heap-size,
                max-temp-stk-size,
                max-ctrl-stk-size,
                word-size)
=   (p-clock1 (t,
               s->lr (logic->s (expr, alist, program-names),
                      heap-size,
                      max-temp-stk-size,
                      max-ctrl-stk-size,
                      word-size),
               cdr (v&c$ (t, expr, alist)) + 1)
    +   2)
```

DEFINITION:
total-heap-reqs (*expr*, *alist*, *program-names*, *heap-size*)
= lr-total-heap-reqs (*expr*,
alist,
program-names,
heap-size,
cdr (v&c$ (**t**, *expr*, *alist*)) + 1)

DEFINITION:
max-ctrl-reqs (*expr*, *alist*, *program-names*)
= lr-max-ctrl-reqs (*expr*,
alist,
program-names,
cdr (v&c$ (**t**, *expr*, *alist*)) + 1)

DEFINITION:
max-temp-reqs (*expr*, *alist*, *program-names*)
= lr-max-temp-reqs (*expr*,
alist,
program-names,
cdr (v&c$ (**t**, *expr*, *alist*)) + 1)

DEFINITION:
max-word-size-reqs (*expr*, *alist*, *program-names*, *heap-size*)
= lr-max-word-size-reqs (*expr*,
alist,
program-names,
heap-size,
cdr (v&c$ (**t**, *expr*, *alist*)) + 1)

THEOREM: s-formals-s-prog-logic->s
s-formals (s-prog (logic->s (*expr*, *alist*, *pnames*))) = strip-cars (*alist*)

THEOREM: deposit-answer-addr-preserves-lr-valp
(adpp (untag (LR-ANSWER-ADDR), *data-seg*) ∧ lr-valp (*value*, *addr*, *data-seg*))
→ lr-valp (*value*,
addr,
deposit (*anything*, identity (LR-ANSWER-ADDR), *data-seg*))

THEOREM: p-last-2-instrs-main-program
(adpp (untag (LR-ANSWER-ADDR), p-data-segment (*l*))
∧ (p-psw (*l*) = 'run)
∧ listp (p-temp-stk (*l*))
∧ lr-valp (*value*, car (p-temp-stk (*l*)), p-data-segment (*l*)))
→ lr-valp (*value*,

fetch (identity (LR-ANSWER-ADDR),
p-data-segment (p (p-set-pc (lr->p (*l*),
tag ('pc,
cons (name (car (p-prog-segment (*l*))),
lr-p-c-size (**t**,
program-body (car (p-prog-segment (*l*))))
2))),
p-data-segment (p (p-set-pc (lr->p (*l*),
tag ('pc,
cons (name (car (p-prog-segment (*l*))),
lr-p-c-size (**t**,
program-body (car (p-prog-segment (*l*)))))))),
2)))

THEOREM: lr-programs-properp-s->lr-logic->s
(l-proper-exprp (**t**, *expr*, *pnames*, strip-cars (*alist*))
∧ l-proper-programsp (*pnames*)
∧ all-litatoms (strip-cars (*alist*))
∧ l-data-seg-body-restrictedp (**t**, *expr*)
∧ l-restrict-subrps-progs (*pnames*)
∧ l-restrict-subrps (**t**, *expr*)
∧ l-restrictedp (*pnames*, *alist*)
∧ (*heap-size* ≮ s-total-heap-reqs (s-progs (logic->s (*expr*, *alist*, *pnames*)),
alist,
heap-size)))
→ lr-programs-properp (s->lr (logic->s (*expr*, *alist*, *pnames*),
heap-size,
max-ctrl,
max-temp,
word-size),
cdr (lr-data-seg-table (s-progs (logic->s (*expr*,
alist,
pnames)),
alist,
heap-size)))

THEOREM: logic->p-ok-really-helper-1
(proper-p-statep (lr->p (*l*))
∧ good-posp1 (offset (p-pc (*l*)), program-body (p-current-program (*l*)))
∧ lr-programs-properp (*l*, *table*)
∧ lr-p-proper-statep (p-temp-stk (*l*),
p-ctrl-stk (*l*),
p-data-segment (*l*),
table)

$\wedge$ lr-proper-formalsp (cdr (p-prog-segment (l)))
$\wedge$ (p-psw (lr-eval (**t**, l, c)) = 'run)
$\wedge$ adpp (untag (LR-ANSWER-ADDR), p-data-segment (l))
$\wedge$ (area-name (p-pc (l)) = name (car (p-prog-segment (l))))
$\wedge$ (offset (p-pc (l)) = **nil**)
$\wedge$ lr-valp (*value*,
car (p-temp-stk (lr-eval (**t**, l, c))),
p-data-segment (lr-eval (**t**, l, c))))
$\rightarrow$ lr-valp (*value*,
fetch (LR-ANSWER-ADDR,
p-data-segment (p (p (lr->p (l), p-clock1 (**t**, l, c)), 2))),
p-data-segment (p (p (lr->p (l), p-clock1 (**t**, l, c)), 2)))

THEOREM: name-car-p-prog-segment-s->lr
name (car (p-prog-segment (s->lr (s, *heap-size*, *max-ctrl*, *max-temp*, *word-size*))))
= name (car (s-progs (s)))

THEOREM: p-pc-s->lr
p-pc (s->lr (s, *heap-size*, *max-temp*, *max-ctrl*, *word-size*))
= tag ('pc, cons (s-pname (s), s-pos (s)))

THEOREM: adpp-untag-lr-answer-addr-s->lr
adpp (identity (untag (LR-ANSWER-ADDR)),
p-data-segment (s->lr (s, *heap-size*, *max-ctrl*, *max-temp*, *word-size*)))

THEOREM: lr-s-similar-statesp-s->lr-logic->s-lr-data-seg-table
((*heap-size* $\not<$ s-total-heap-reqs (s-progs (logic->s (*expr*, *alist*, *pnames*)),
alist,
heap-size))
$\wedge$ l-restrictedp (*pnames*, *alist*)
$\wedge$ l-data-seg-body-restrictedp (**t**, *expr*))
$\rightarrow$ lr-s-similar-statesp (*alist*,
nil,
s->lr (logic->s (*expr*, *alist*, *pnames*),
heap-size,
max-ctrl,
max-temp,
word-size),
cdr (lr-data-seg-table (s-progs (logic->s (*expr*,
alist,
pnames)),
alist,
heap-size)))

THEOREM: s-eval-flag-run-v&c$-not-f-flag-t

(v&c$ (**t**, *expr*, *alist*)
 ∧ (cdr (v&c$ (**t**, *expr*, *alist*)) < *c*)
 ∧ l-proper-programsp (*pnames*)
 ∧ l-proper-exprp (**t**, *expr*, *pnames*, strip-cars (*alist*))
 ∧ all-litatoms (strip-cars (*alist*)))
→ (s-err-flag (s-eval (**t**, logic->s (*expr*, *alist*, *pnames*), *c*)) = '`run`)

THEOREM: lr-proper-formalsp-cdr-p-prog-segment-s->lr-logic->s
(l-proper-programsp (*pnames*)
 ∧ l-proper-exprp (**t**, *expr*, *pnames*, strip-cars (*alist*))
 ∧ all-litatoms (strip-cars (*alist*)))
→ lr-proper-formalsp (cdr (p-prog-segment (s->lr (logic->s (*expr*,
alist,
pnames),
heap-size,
max-ctrl,
max-temp,
word-size))))

THEOREM: lr-s-similar-params-lr-good-pointerps-strip-cdrs
lr-s-similar-params (*s-params*, *lr-params*, *data-seg*)
→ lr-good-pointerps (strip-cdrs (*lr-params*), *data-seg*)

THEOREM: lr-p-proper-statep-s->lr
((*heap-size* ≮ s-total-heap-reqs (s-progs (*s*), s-params (*s*), *heap-size*))
 ∧ s-restrictedp (s-progs (*s*), s-params (*s*))
 ∧ (*params* = s-params (*s*)))
→ lr-p-proper-statep (p-temp-stk (s->lr (*s*,
heap-size,
max-ctrl,
max-temp,
word-size)),
p-ctrl-stk (s->lr (*s*,
heap-size,
max-ctrl,
max-temp,
word-size)),
p-data-segment (s->lr (*s*,
heap-size,
max-ctrl,
max-temp,
word-size)),
cdr (lr-data-seg-table (s-progs (*s*), *params*, *heap-size*)))

THEOREM: proper-p-statep-lr->p-s->lr-logic->s

$$
\begin{array}{ll}
 & (\text{l-proper-exprp}\,(\mathbf{t},\ \textit{expr},\ \textit{pnames},\ \text{strip-cars}\,(\textit{alist})) \\
\wedge & \quad \text{l-proper-programsp}\,(\textit{pnames}) \\
\wedge & \quad \text{all-litatoms}\,(\text{strip-cars}\,(\textit{alist})) \\
\wedge & \quad \text{l-data-seg-body-restrictedp}\,(\mathbf{t},\ \textit{expr}) \\
\wedge & \quad \text{l-restrictedp}\,(\textit{pnames},\ \textit{alist}) \\
\wedge & \quad \text{l-restrict-subrps}\,(\mathbf{t},\ \textit{expr}) \\
\wedge & \quad \text{l-restrict-subrps-progs}\,(\textit{pnames}) \\
\wedge & \quad (\textit{heap-size} \not< (\text{s-total-heap-reqs}\,(\text{s-progs}\,(\text{logic->s}\,(\textit{expr},\ \textit{alist},\ \textit{pnames})), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{alist}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{heap-size}) \\
 & \qquad\qquad\qquad + \quad \text{s-eval-heap-r}\,(\mathbf{t},\ \text{logic->s}\,(\textit{expr},\ \textit{alist},\ \textit{pnames}),\ \textit{c}))) \\
\wedge & \quad (\textit{max-ctrl} \not< \text{lr-max-ctrl-reqs}\,(\textit{expr},\ \textit{alist},\ \textit{pnames},\ \textit{c})) \\
\wedge & \quad (\textit{max-ctrl} < \exp\,(2,\ \textit{word-size})) \\
\wedge & \quad (\textit{max-ctrl} \in \mathbf{N}) \\
\wedge & \quad (\textit{max-temp} \not< \text{lr-max-temp-reqs}\,(\textit{expr},\ \textit{alist},\ \textit{pnames},\ \textit{c})) \\
\wedge & \quad (\textit{max-temp} < \exp\,(2,\ \textit{word-size})) \\
\wedge & \quad (\textit{max-temp} \in \mathbf{N}) \\
\wedge & \quad (\textit{word-size} \not< \text{lr-max-word-size-reqs}\,(\textit{expr},\ \textit{alist},\ \textit{pnames},\ \textit{heap-size},\ \textit{c})) \\
\wedge & \quad (\textit{word-size} \in \mathbf{N})) \\
\rightarrow & \quad \text{proper-p-statep}\,(\text{lr->p}\,(\text{s->lr}\,(\text{logic->s}\,(\textit{expr},\ \textit{alist},\ \textit{pnames}), \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{heap-size}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{max-ctrl}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{max-temp}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{word-size})))
\end{array}
$$

THEOREM: logic->p-ok-really

$$
\begin{array}{ll}
 & (\text{l-proper-exprp}\,(\mathbf{t},\ \textit{expr},\ \textit{pnames},\ \text{strip-cars}\,(\textit{alist})) \\
\wedge & \quad \text{l-proper-programsp}\,(\textit{pnames}) \\
\wedge & \quad \text{all-litatoms}\,(\text{strip-cars}\,(\textit{alist})) \\
\wedge & \quad \text{l-data-seg-body-restrictedp}\,(\mathbf{t},\ \textit{expr}) \\
\wedge & \quad \text{l-restrictedp}\,(\textit{pnames},\ \textit{alist}) \\
\wedge & \quad \text{v\&c\$}\,(\mathbf{t},\ \textit{expr},\ \textit{alist}) \\
\wedge & \quad \text{l-restrict-subrps}\,(\mathbf{t},\ \textit{expr}) \\
\wedge & \quad \text{l-restrict-subrps-progs}\,(\textit{pnames}) \\
\wedge & \quad (\textit{heap-size} \not< \text{total-heap-reqs}\,(\textit{expr},\ \textit{alist},\ \textit{pnames},\ \textit{heap-size})) \\
\wedge & \quad (\textit{max-ctrl} \not< \text{max-ctrl-reqs}\,(\textit{expr},\ \textit{alist},\ \textit{pnames})) \\
\wedge & \quad (\textit{max-ctrl} < \exp\,(2,\ \textit{word-size})) \\
\wedge & \quad (\textit{max-ctrl} \in \mathbf{N}) \\
\wedge & \quad (\textit{max-temp} \not< \text{max-temp-reqs}\,(\textit{expr},\ \textit{alist},\ \textit{pnames})) \\
\wedge & \quad (\textit{max-temp} < \exp\,(2,\ \textit{word-size})) \\
\wedge & \quad (\textit{max-temp} \in \mathbf{N}) \\
\wedge & \quad (\textit{word-size} \not< \text{max-word-size-reqs}\,(\textit{expr},\ \textit{alist},\ \textit{pnames},\ \textit{heap-size})) \\
\wedge & \quad (\textit{word-size} \in \mathbf{N})) \\
\rightarrow & \quad \text{lr-valp}\,(\text{car}\,(\text{v\&c\$}\,(\mathbf{t},\ \textit{expr},\ \textit{alist})),
\end{array}
$$

```
fetch (LR-ANSWER-ADDR,
       p-data-segment (p (logic->p (expr,
                                    alist,
                                    pnames,
                                    heap-size,
                                    max-ctrl,
                                    max-temp,
                                    word-size),
                          logic->p-clock (expr,
                                          alist,
                                          pnames,
                                          heap-size,
                                          max-ctrl,
                                          max-temp,
                                          word-size)))),
p-data-segment (p (logic->p (expr,
                             alist,
                             pnames,
                             heap-size,
                             max-ctrl,
                             max-temp,
                             word-size),
                   logic->p-clock (expr,
                                   alist,
                                   pnames,
                                   heap-size,
                                   max-ctrl,
                                   max-temp,
                                   word-size))))
```

Index