```
(defunc get-vars-t (px acc)
  :ic (and (PropEx px) (lovp acc))
  :oc (lovp (get-var-t px acc))
  (cond ((booleanp px) acc)
        ((varp px) (add px acc))
        ((UnaryExp px)
                (get-vars-t (second px) acc))
        (t      (get-vars-t (first px) (get-vars-t (third px) acc)))))
```

if not in then cons

```
(defdata PropEx (oneof boolean
                       var UnExp BinExp))
(defdata UnEx (list UnaryOp PropEx))
(defdata BinEx (list PropEx BinOp PropEx))

(defdata var (oneof 'a 'b 'c 'q 's 'p 'r))
```

' ∧ ∨ ⊕ =

¬((P∧q)⊕(q∨r))

```
(defunc get-vars (px)
  :ic (PropExp px)
  :oc (lovp (get-vars px))
  (cond ((boolexp px) ())
        ((varp px) (list px))
        ((Unexp px) (get-vars (second px)))
        (t (combine (get-vars (first px))
                    (get-vars (third px))))))
```

```
(defunc get-vars* (px)
  :ic (PropExp* px)
  :oc (lovp (get-vars* px))
  (get-vars-t px nil))
```

```
(defunc combine (x y)
  ;ic (and (listp x) (listp y))
  :oc (listp (combine x y))
  (if (endp x)
      y
      (add (first x) (combine (rest x) y))))
```

add-nodups

C1. (PropExp px)

(get-vars* px)

$\equiv$ {Def get-vars*}

(get-vars-t px nil)

$\equiv$ {L1 ??}

(combine (getvars px) nil)

$\equiv$ {$L2_{\text{combine px nil}}$}

(getvars px)

((p ∧ q) ⊕ (q ∨ r))

	(get-vars-t)	acc	(get-vars px)	px
0				
1	'(p q r)	'(q r)	'(p q)	(p ∧ q)
	(p q r)	'(q r)	'(p)	p

L1: (PropExp px) ∧ (lovp acc) = (get-vars-t px acc)

= (combine (getvars px) acc)

I.S. to prove $L1$ using got-vars-t

1) $\neg IC \Rightarrow L1$

2) $IC \wedge (\text{boolexp } px) \Rightarrow L1$

3) $IC \wedge \neg(\text{boolexp } px) \wedge (\text{varp } px) \Rightarrow L1$

4) $IC \wedge \neg(\text{boolexp } px) \wedge \neg(\text{varp } px) \wedge (\text{Unexp } px) \wedge L1|_{((px\ (\text{second } px))} \Rightarrow L1$

5) $IC \wedge \neg(\text{boolexp } px) \wedge \neg(\text{varp } px) \wedge \neg(\text{Unexp } px)$

$\wedge L1|_{((px\ (\text{second } px))}$

$(acc\ (\text{got-vars-t } (\text{third } px)\ acc))$

$\wedge L1|_{((px\ (\text{third } px))} \Rightarrow L1$

$(\text{Listp } l) \Rightarrow (\text{perm } l\ (\text{rev } l))$

$(\text{perm } l\ (\text{rev } l))$

$\equiv (\text{perm } l\ (\text{app}(\text{rev}(\text{rest } l))(\text{list}(\text{first } l))))$

$\equiv (\text{in}(\text{first } l)(\text{app}(\text{rev}(\text{rest } l))(\text{list}(\text{first } l)))) \wedge (\text{perm}(\text{rest } l)(\text{del}(\text{first } l)$

$(\text{app} \text{ ——})$

$l = (a\ b\ a)$

```
(defunc render scan (x y)
  :ic (cond (natp x)(natp y))
  :oc (lonp (scan x y))
  (cond ((and (>= x 1024)(>= y 1024)) nil)
        ((>= x 1024)(cons (* x y) (scan 0 (+ y 1))))
        (t          (scan (+ x 1) y))))
```

pretend never 1024

```
(defunc m.scan (x y)
  :ic (and (natp x) (natp y))
  :oc (natp (m.scan x y))
(* 1024 (abs (1024 - y))) + (abs (1024 - x)))
```

Oblig 1

$IC \wedge \neg((x \geq 1024) \wedge (y \geq 1024)) \wedge (x \geq 1024)$

$\Rightarrow (m.scan\ x\ y) > (m.scan\ 0\ (+\ y\ 1))$

Obligation 2

$IC \wedge \neg((x \geq 1024) \wedge (y \geq 1024)) \wedge$

$\neg(x > 1024) \Rightarrow (m.scan\ x\ y) > (m.scan\ (+\ x\ 1)\ y)$

```
(defunc del* (e x)
  :ic (listp x)
  :oc (listp (del* e x))
(rev (del-t e x nil)))

(defunc del-t (e x acc)
  :ic (and (listp x) (listp acc))
  :oc (listp (del-t e x acc))
 (cond ((endp x) acc)
       ((equal e (first x)) (app (rev (rest x)) acc))
       (t (del-t e (rest x) (cons (first x) acc)))))
```

```
(defunc del (e x)
  :ic (listp x)
  :oc (listp (del e x))
 (cond ((endp x) nil)
       ((equal e (first x)) (rest x))
       (t (cons (first x) (del e (rest x))))))
```

C1. (listp x)

(del * e x)

= { Def del* }

(rev (del-t e x nil))

≡ { C1 }

(rev (app (rev (del e x)) nil))

$\equiv \{ \varphi_{app.x\,nil} \}$

(rev (rev (del e x)))

$\equiv \{ \varphi_{rev\text{-}rev} \}$

(del e x)

e	l	acc	del-t	del
c	'(a b c d e)	nil	'(e d b a)	'(a b d e)
	(b c d e)	'(a)	(e d b a)	(b d e)
	(c d e)	'(b a)	.. ..	(d e)

IS for del-t

1) $\neg IC \Rightarrow \varphi$
2) $IC \wedge (endp\ x) \Rightarrow \varphi$
3) $IC \wedge \neg(endp\ x) \wedge (e = (first\ x)) \Rightarrow \varphi$
4) $IC \wedge \neg(endp\ x) \wedge \neg(e = (first\ x)) \wedge \varphi|_{((x\ (rest\ x))\ (acc\ (cons\ (first\ x)\ acc)))} \Rightarrow \varphi$

L1: $(listp\ x) \wedge (listp\ acc) \Rightarrow [(del\text{-}t\ e\ x\ acc) = (app\ (rev\ (del\ e\ x))\ acc)]$

Obligation 3

C1. (listp r)
C2. (listp acc)
C3 (endp x)
C4. (e = (first x))

(del-t e x acc)
≡ {C3, C4, Def del-t}
(app (rev (rest x)) acc)
≡

RHS
(app (rev (del e x)) acc)
≡ {Def del, C4, C3}
(app (rev (rest x)) acc)

LHS = RHS

Obligation 4

C1. (listp x)
C2. (listp acc)
C3. ¬(endp x)
C4. (e ≠ (first x))
C5. (listp (rest x)) ∧ (listp (cons (first x) acc))
    ⇒ ((del-t (rest x) (cons (first x) acc))
    = (app (rev (del e (rest x))) (cons (first x) acc)))
C6. (listp (rest x)) {C1, C3, Def listp}
C7. (listp cacc) {def listp, C2}
C8. α {C5, C6, C7, MP}

rx
cacc
α

LHS (del-t e x acc)

$\equiv$ {C4, C3, Def del-t}

(del-t e rx cacc)

$\equiv$ {C8}

(app (rev (dele rx)) cacc)

RHS (app (rev (dele x)) acc)

$\equiv$ {C4, C3, Def del}

(app (rev (cons (first x) (dele (rest x)))) acc)

$\equiv$ {first-rest axioms, def rev, def endp}

(app (app (rev (dele (rest x))) (list (first x))) acc)

$\equiv$ {Assoc of app} Magic

(app (rev (dele (rest x))) (cons (first x) acc))

```
(defunc f (x y)
  :ic (and (listp x) (integerp y))
  :oc t
  (cond ((equal y 0) x)
        ((endp x) x)
        (+ (f (rest x) (+ y 1)))))
```

```
(defunc m (x y)
  :ic
  :oc (natp (m x y))
  (len x))
```

(integerp x) ∧ (posp x) ∧ (x > y) ⇒ (m x y) > (m (+ x y) y)

```
(defunc f (x y)
  :ic (and (integerp x) (posp y))
  :oc t
  (if (> x y)
      y
      (f (+ x y) y)))
```

```
(defunc m (x y)
  :ic (and (integerp x) (posp y))
  :oc (natp (m x y))
  (+ 1 (- y x)))
```

```
(if (> x y)
    0
    (+ (- y x) y))
```

```
(defunc replicate (l)
  :ic (listp l)
  :oc (listp (replicate l))
 (if (endp l)
     l
     (cons (list (first l) (first l)) (replicate (rest l)))))
```

```
(defunc replicate* (l)
   :ic (listp l)
   :oc (listp (replicate* l))
   (rev (replicate-t l nil)))
```

(replicate '(2 3)) = '((2 2) (3 3))

```
(defunc replicate-t (l acc)
   :ic (and (listp l) (listp acc))
   :oc (listp (replicate-t l acc))
   (if (endp l)
       acc
       (replicate-t (rest l) (cons (list (first l) (first l)) acc))))
```

(replicate-t '(2 3) nil) = '((3 3) (2 2))

(replicate-t '(2 3) '((0 0) (1 1))) = '((3 3) (2 2) (0 0) (1 1))
= (replicate-t '(1 0 2 3))

L1: $(listp\ l) \wedge (listp\ acc) \Rightarrow ((replicate\text{-}t\ l\ acc) = (app\ (rev\ (replicate\ l))\ acc))$

Prove replicate* = replicate

cl. $(listp\ l)$

$(replicate^*\ l)$

$= \{C1, Def\ replicate^*\}$

$(rev\ (replicate\text{-}t\ l\ nil))$

$= \{L1\}$

$(rev\ (app\ (rev\ (replicate\ l))\ nil))$

$= \{\phi_{app\_x\_nil}\}$

$(rev\ (rev\ (replicate\ l)))$

$= \{\phi_{rev\_rev}\}$

$(replicate\ l)$

Prove L1

I.S for replicate-t

1) ¬((listp l) ∧ (listp acc)) ⇒ L1

2) (listp l) ∧ (listp acc) ∧ (endp l) ⇒ L1

*3) (listp l) ∧ (listp acc) ∧ ¬(endp l)

∧ L1 |((l (rest l))
(acc (cons (list (first l) (first l)) acc)))

⇒ L1

C1. (listp l)

C2. (listp acc)

C3. ¬(endp l)

C4. (listp (rest l)) ∧ (listp cacc) ⇒ (replicate-t (rest l) cacc) = (app (rev (replicate (rest l))) cacc)

C5. (listp (rest x)) {C1, C3, Def listp}

C6. (listp cacc) {C2, Def listp}

C7. (replicate-t (rest l) cacc)
= (app (rev (replicate (rest l))) cacc)
{C4, C5, C6, MP}

(app (rev (replicate l)) acc)

≡ {Def replicate, (3)}

(app (rev (cons (list (first l) (first l)) (replicate (rest l)))) acc)

≡ {Def rev, first-rest axiom}

(app (app (rev (replicate (rest l))) (list (list (first l) (first l)))) acc)

≡ {Assoc of app}

(app (rev (replicate (rest l))) (app (list (list (first l) (first l))) acc))

≡ {Def list, def endp, def app}

(app (rev (replicate (rest l))) (cons (first l) (first l) acc))

≡ {C7}

(app (rev (replicate (rest l))) (cons (first l) (first l) acc))

≡ {Def replicate-t, (3)}

(replicate-t (rest l) (cons (first l) (first l) acc))

≡ {Def replicate-t, (3)}

(replicate l acc)

QED