```
#|
```



```
|#
```

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

```
;Annotated script for mechanical proof of the Tautology theorem.
;Proof involves -
;Definition of proof-checker for Schoenfield's FOL.
;Proof of several derived inference rules, primarily the
;subset lemma.
;Definition of tautology-checker.
;Every tautology has a proof.
;Correctness of tautology-checker - every tautology is
;always logically-true, and all logical-truths are tautologies.
;First, functions, variables and predicate symbols.
```

DEFINITION:
function (fn)
$= \quad ((fn = \text{list}(\texttt{'f}, \text{cadr}(fn), \text{caddr}(fn)))$
$\quad \wedge \quad (\text{cadr}(fn) \in \mathbf{N})$
$\quad \wedge \quad (\text{caddr}(fn) \in \mathbf{N}))$

DEFINITION:
variable $(x) = ((x = \text{list}(\texttt{'x}, \text{cadr}(x))) \land (\text{cadr}(x) \in \mathbf{N}))$

DEFINITION:
predicate (p)
$= \quad (((p = \text{list}(\texttt{'p}, \text{cadr}(p), \text{caddr}(p)))$
$\quad \land \quad (\text{cadr}(p) \in \mathbf{N})$
$\quad \land \quad (\text{caddr}(p) \in \mathbf{N}))$
$\quad \lor \quad (p = \texttt{'equal}))$

DEFINITION:
degree (fn)
$=$ **if** $fn = \texttt{'equal}$ **then** 2
else caddr (fn) **endif**

DEFINITION: index $(fn) = \text{cadr}(fn)$

DEFINITION: func-pred $(x) = (\text{function}(x) \lor \text{predicate}(x))$

DEFINITION: v $(x) = \text{list}(\texttt{'x}, \text{fix}(x))$

THEOREM: numberp-fix
$\text{fix}(x) \in \mathbf{N}$

THEOREM: variable-v
variable $(\text{v}(x))$

DEFINITION: fn $(x, y) = \text{list}(\texttt{'f}, \text{fix}(x), \text{fix}(y))$

DEFINITION: p $(x, y) = \text{list}(\texttt{'p}, \text{fix}(x), \text{fix}(y))$

THEOREM: function-fn
function $(\text{fn}(x, y))$

THEOREM: predicate-p
predicate $(\text{p}(x, y))$

`;quantifer, there exists.`

DEFINITION: quantifier $(x) = (x = \texttt{'forsome})$

DEFINITION:
$(x \cup y)$
$=$ **if** listp (x)
then if $\text{car}(x) \in y$ **then** $\text{cdr}(x) \cup y$
else $\text{cons}(\text{car}(x), \text{cdr}(x) \cup y)$ **endif**
else y **endif**

EVENT: Enable variable; name this event ‘g0223’.

EVENT: Enable quantifier; name this event ‘g0224’.

THEOREM: predicate-f-equal
predicate (`'equal`)

EVENT: Enable function; name this event ‘g0225’.

EVENT: Enable predicate; name this event ‘g0226’.

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

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

THEOREM: not-member-delete
$x \notin$ delete (x, y)

```
;returns list of free variables in EXP.
```

DEFINITION:
collect-free (*exp*, *flg*)
= **if** listp (*exp*)
then if *flg* = **t**
then if variable (*exp*) **then** cons (*exp*, **nil**)
elseif quantifier (car (*exp*)) ∧ listp (cdr (*exp*))
then delete (cadr (*exp*), collect-free (cddr (*exp*), `'list`))
elseif func-pred (car (*exp*))
∨ (car (*exp*) = `'not`)
∨ (car (*exp*) = `'or`)
then collect-free (cdr (*exp*), `'list`)
else nil endif
else append (collect-free (car (*exp*), **t**),
collect-free (cdr (*exp*), `'list`)) **endif**
else nil endif

DEFINITION: sentence (*exp*) = (collect-free (*exp*, **t**) = **nil**)

```
;returns bound variables in EXP that surround free occurrences of VAR.
```

DEFINITION:
covering (*exp*, *var*, *flg*)
= **if** listp (*exp*)
  then if *flg* = **t**
    then if variable (*exp*) **then nil**
      elseif quantifier (car (*exp*)) ∧ listp (cdr (*exp*))
      then if cadr (*exp*) = *var* **then nil**
        elseif *var* ∈ collect-free (cddr (*exp*), `'list`)
        then cons (cadr (*exp*), covering (cddr (*exp*), *var*, `'list`))
        else nil endif
      elseif func-pred (car (*exp*))
        ∨ (car (*exp*) = `'not`)
        ∨ (car (*exp*) = `'or`)
      then covering (cdr (*exp*), *var*, `'list`)
      else nil endif
    else append (covering (car (*exp*), *var*, **t**),
      covering (cdr (*exp*), *var*, `'list`)) **endif**
  else nil endif

```
;X and Y are disjoint.
```

DEFINITION:
nil-intersect (x, y)
= **if** listp (x) **then** (car (x) $\notin$ y) ∧ nil-intersect (cdr (x), y)
  else t endif

```
;TERM is free for VAR in EXP.
```

DEFINITION:
free-for (*exp*, *var*, *term*, *flg*)
= nil-intersect (covering (*exp*, *var*, *flg*), collect-free (*term*, **t**))

DEFINITION: f-equal (x, y) = list (`'equal`, x, y)

DEFINITION: f-not (x) = list (`'not`, x)

DEFINITION: f-or (x, y) = list (`'or`, x, y)

DEFINITION: forsome (x, y) = list (`'forsome`, x, y)

DEFINITION: f-and (x, y) = f-not (f-or (f-not (x), f-not (y)))

DEFINITION: $\text{f-implies}(x, y) = \text{f-or}(\text{f-not}(x), y)$

DEFINITION: $\forall\ var\ exp = \text{f-not}(\text{forsome}(var, \text{f-not}(exp)))$

DEFINITION: $\text{f-iff}(x, y) = \text{f-and}(\text{f-implies}(x, y), \text{f-implies}(y, x))$

DEFINITION:
var-list (*list*, *n*)
= **if** $n \simeq 0$ **then** $list =$ **nil**
else variable (car (*list*)) $\wedge$ var-list (cdr (*list*), $n - 1$) **endif**

DEFINITION:
var-set (*list*, *n*)
= **if** $n \simeq 0$ **then** $list =$ **nil**
else variable (car (*list*))
$\wedge$ (car (*list*) $\notin$ cdr (*list*))
$\wedge$ var-set (cdr (*list*), $n - 1$) **endif**

```
;Recognizer for terms.
```

DEFINITION:
termp (*exp*, *flg*, *count*)
= **if** *flg* = **t**
then if $exp \simeq$ **nil** **then f**
else variable (*exp*)
$\vee$ (function (car (*exp*))
$\wedge$ termp (cdr (*exp*),
`'list`,
degree (car (*exp*)))) **endif**
elseif ($exp \simeq$ **nil**) $\vee$ ($count \simeq 0$) **then** ($exp =$ **nil**) $\wedge$ ($count \simeq 0$)
else termp (car (*exp*), **t**, 0) $\wedge$ termp (cdr (*exp*), `'list`, $count - 1$) **endif**

DEFINITION: $\text{arg1}(x) = \text{cadr}(x)$

DEFINITION: $\text{arg2}(x) = \text{caddr}(x)$

```
;EXP is an atom, pred. symbol followed by list of terms.
```

DEFINITION:
atomp (*exp*)
= (predicate (car (*exp*)) $\wedge$ termp (cdr (*exp*), `'list`, degree (car (*exp*))))

EVENT: Enable atomp; name this event 'g0253'.

```
;EXP is a formula
```

Definition:
formula (*exp*, *flg*, *count*)

```
=   if flg = t
    then if exp ≃ nil then f
         else atomp (exp)
              ∨  ((car (exp) = 'not)
                  ∧  formula (cdr (exp), 'list, 1))
              ∨  ((car (exp) = 'or)
                  ∧  formula (cdr (exp), 'list, 2))
              ∨  ((car (exp) = 'forsome)
                  ∧  variable (cadr (exp))
                  ∧  formula (cddr (exp), 'list, 1)) endif
    elseif (exp ≃ nil) ∨ (count ≃ 0) then (exp = nil) ∧ (count ≃ 0)
    else formula (car (exp), t, 0)
         ∧  formula (cdr (exp), 'list, count − 1) endif
```

```
;Result of substituting TERM for VAR in EXP.
```

Definition:
subst (*exp*, *var*, *term*, *flg*)

```
=   if listp (exp)
    then if flg = t
         then if variable (exp)
              then if exp = var then term
                   else exp endif
              elseif quantifier (car (exp)) ∧ listp (cdr (exp))
              then if cadr (exp) = var then exp
                   else cons (car (exp),
                              cons (cadr (exp),
                                    subst (cddr (exp), var, term, 'list))) endif
              elseif func-pred (car (exp))
                     ∨  (car (exp) = 'not)
                     ∨  (car (exp) = 'or)
              then cons (car (exp), subst (cdr (exp), var, term, 'list))
              else exp endif
         else cons (subst (car (exp), var, term, t),
                    subst (cdr (exp), var, term, 'list)) endif
    else exp endif
```

Definition:
neg (*exp1*, *exp2*) = ((*exp1* = f-not (*exp2*)) ∨ (*exp2* = f-not (*exp1*)))

Definition:
conc (*pf*, *flg*)

```
=   if pf ≃ nil then nil
```

```
    elseif flg = t then caddr (pf)
    else cons (conc (car (pf), t), conc (cdr (pf), 'list)) endif
```

DEFINITION:
subset (*x*, *y*)
= **if** listp (*x*) **then** (car (*x*) $\in$ *y*) $\wedge$ subset (cdr (*x*), *y*)
else t endif

DEFINITION: set-equal (*x*, *y*) = (subset (*x*, *y*) $\wedge$ subset (*y*, *x*))

`;The axioms: propositional, substitution, identity, equality for functions and predicates.`

DEFINITION: prop-axiom (*exp*) = f-or (f-not (*exp*), *exp*)

DEFINITION:
subst-axiom (*exp*, *var*, *term*)
= f-implies (subst (*exp*, *var*, *term*, **t**), forsome (*var*, *exp*))

DEFINITION: ident-axiom (*var*) = f-equal (*var*, *var*)

DEFINITION:
pairequals (*vars1*, *vars2*, *exp*)
= **if** listp (*vars1*)
then f-implies (f-equal (car (*vars1*), car (*vars2*)),
pairequals (cdr (*vars1*), cdr (*vars2*), *exp*))
else *exp* **endif**

DEFINITION:
equal-axiom2 (*vars1*, *vars2*, *pr*)
= pairequals (*vars1*, *vars2*, f-implies (cons (*pr*, *vars1*), cons (*pr*, *vars2*)))

DEFINITION:
assume (*exp*, *list*, *flg*)
= **if** listp (*list*)
then if (caaar (*list*) = *flg*) $\wedge$ (*exp* = cadar (*list*))
then cdr (*list*)
else assume (*exp*, cdr (*list*), *flg*) **endif**
else f endif

`;Proof-constructors`

DEFINITION:
prop-axiom-proof (*exp*)
= list (`'axiom`, list (`'prop-axiom`, *exp*), prop-axiom (*exp*))

DEFINITION:
subst-axiom-proof (*exp*, *var*, *term*)
= list (`'axiom`,
list (`'subst-axiom`, *exp*, *var*, *term*),
subst-axiom (*exp*, *var*, *term*))

DEFINITION:
ident-axiom-proof (*var*)
= list (`'axiom`, list (`'ident-axiom`, *var*), f-equal (*var*, *var*))

DEFINITION:
equal-axiom1 (*vars1*, *vars2*, *fn*)
= pairequals (*vars1*, *vars2*, f-equal (cons (*fn*, *vars1*), cons (*fn*, *vars2*)))

DEFINITION:
equal-axiom1-proof (*fn*, *vars1*, *vars2*)
= list (`'axiom`,
list (`'equal-axiom1`, *fn*, *vars1*, *vars2*),
equal-axiom1 (*vars1*, *vars2*, *fn*))

DEFINITION:
equal-axiom2-proof (*pr*, *vars1*, *vars2*)
= list (`'axiom`,
list (`'equal-axiom2`, *pr*, *vars1*, *vars2*),
equal-axiom2 (*vars1*, *vars2*, *pr*))

DEFINITION:
expan-proof (*a*, *b*, *pf*) = list (`'rule`, list (`'expan`, *a*, *b*), f-or (*a*, *b*), *pf*)

DEFINITION:
contrac-proof (*a*, *pf*) = list (`'rule`, list (`'contrac`, *a*), *a*, *pf*)

DEFINITION:
assoc-proof (*a*, *b*, *c*, *pf*)
= list (`'rule`, list (`'assoc`, *a*, *b*, *c*), f-or (f-or (*a*, *b*), *c*), *pf*)

DEFINITION:
cut-proof (*a*, *b*, *c*, *pf1*, *pf2*)
= list (`'rule`, list (`'cut`, *a*, *b*, *c*), f-or (*b*, *c*), list (*pf1*, *pf2*))

DEFINITION:
forsome-intro-proof (*var*, *a*, *b*, *pf*)
= list (`'rule`, list (`'e-intro`, *var*, *a*, *b*), f-implies (forsome (*var*, *a*), *b*), *pf*)

EVENT: Disable atomp; name this event ‘g2737’.

DEFINITION: hint1 (*pf*) = caadr (*pf*)

DEFINITION: hint2 (*pf*) = cadadr (*pf*)

DEFINITION: hint3 (*pf*) = caddadr (*pf*)

DEFINITION: hint4 (*pf*) = cadddadr (*pf*)

DEFINITION: sub-proof (*pf*) = cadddr (*pf*)

`;The proof-checker, PF is a proof.`

DEFINITION:
prf (*pf*)
= **if** *pf* ≃ **nil** **then** **f**
  elseif car (*pf*) = `'axiom`
  then **if** hint1 (*pf*) = `'prop-axiom`
    then formula (hint2 (*pf*), **t**, 0)
      ∧ (*pf* = prop-axiom-proof (hint2 (*pf*)))
    elseif hint1 (*pf*) = `'subst-axiom`
    then formula (hint2 (*pf*), **t**, 0)
      ∧ variable (hint3 (*pf*))
      ∧ termp (hint4 (*pf*), **t**, 0)
      ∧ free-for (hint2 (*pf*), hint3 (*pf*), hint4 (*pf*), **t**)
      ∧ (*pf* = subst-axiom-proof (hint2 (*pf*),
        hint3 (*pf*),
        hint4 (*pf*)))
    elseif hint1 (*pf*) = `'ident-axiom`
    then variable (hint2 (*pf*))
      ∧ (*pf* = ident-axiom-proof (hint2 (*pf*)))
    elseif hint1 (*pf*) = `'equal-axiom1`
    then function (hint2 (*pf*))
      ∧ var-list (hint3 (*pf*), degree (hint2 (*pf*)))
      ∧ var-list (hint4 (*pf*), degree (hint2 (*pf*)))
      ∧ (*pf* = equal-axiom1-proof (hint2 (*pf*),
        hint3 (*pf*),
        hint4 (*pf*)))
    elseif hint1 (*pf*) = `'equal-axiom2`
    then predicate (hint2 (*pf*))
      ∧ var-list (hint3 (*pf*), degree (hint2 (*pf*)))
      ∧ var-list (hint4 (*pf*), degree (hint2 (*pf*)))
      ∧ (*pf* = equal-axiom2-proof (hint2 (*pf*),
        hint3 (*pf*),
        hint4 (*pf*)))
    else **f** **endif**

```
elseif car (pf) = 'rule
then if hint1 (pf) = 'expan
     then formula (hint2 (pf), t, 0)
          ∧   (pf = expan-proof (hint2 (pf),
                                 hint3 (pf),
                                 sub-proof (pf)))
          ∧   (conc (sub-proof (pf), t) = hint3 (pf))
          ∧   prf (sub-proof (pf))
     elseif hint1 (pf) = 'contrac
     then (pf = contrac-proof (hint2 (pf), sub-proof (pf)))
          ∧   (conc (sub-proof (pf), t) = f-or (hint2 (pf), hint2 (pf)))
          ∧   prf (sub-proof (pf))
     elseif hint1 (pf) = 'assoc
     then (pf = assoc-proof (hint2 (pf),
                             hint3 (pf),
                             hint4 (pf),
                             sub-proof (pf)))
          ∧   (conc (sub-proof (pf), t)
                =   f-or (hint2 (pf), f-or (hint3 (pf), hint4 (pf))))
          ∧   prf (sub-proof (pf))
     elseif hint1 (pf) = 'cut
     then (pf = cut-proof (hint2 (pf),
                           hint3 (pf),
                           hint4 (pf),
                           car (sub-proof (pf)),
                           cadr (sub-proof (pf))))
          ∧   (conc (sub-proof (pf), 'list)
                =   list (f-or (hint2 (pf), hint3 (pf)),
                              f-or (f-not (hint2 (pf)), hint4 (pf))))
          ∧   prf (car (sub-proof (pf)))
          ∧   prf (cadr (sub-proof (pf)))
     elseif hint1 (pf) = 'e-intro
     then variable (hint2 (pf))
          ∧   (pf = forsome-intro-proof (hint2 (pf),
                                         hint3 (pf),
                                         hint4 (pf),
                                         sub-proof (pf)))
          ∧   (hint2 (pf) ∉ collect-free (hint4 (pf), t))
          ∧   (conc (sub-proof (pf), t)
                =   f-implies (hint3 (pf), hint4 (pf)))
          ∧   prf (sub-proof (pf))
     else f endif
else f endif
```

THEOREM: formula-or-reduc
formula (list ('or, a, b), **t**, 0) = (formula (a, **t**, 0) ∧ formula (b, **t**, 0))

THEOREM: formula-not-reduc
formula (list ('not, a), **t**, 0) = formula (a, **t**, 0)

THEOREM: formula-forsome-reduc
formula (list ('forsome, x, a), **t**, 0) = (variable (x) ∧ formula (a, **t**, 0))

```
;PF is a valid proof of EXP.
```

DEFINITION:
proves (*pf*, *exp*) = ((conc (*pf*, **t**) = *exp*) ∧ formula (*exp*, **t**, 0) ∧ prf (*pf*))

THEOREM: proves-is-formula
proves (*pf*, *exp*) → formula (*exp*, **t**, 0)

THEOREM: proves-is-formula-again
(¬ formula (*exp*, **t**, 0)) → (¬ proves (*pf*, *exp*))

```
;Getting rid of PRF by lemmas.
```

THEOREM: prop-axiom-proves
(formula (*exp*, **t**, 0) ∧ (*concl* = f-or (f-not (*exp*), *exp*)))
→ proves (prop-axiom-proof (*exp*), *concl*)

THEOREM: subst-axiom-proves
(formula (*concl*, **t**, 0)
∧ variable (*var*)
∧ termp (*term*, **t**, 0)
∧ free-for (*exp*, *var*, *term*, **t**)
∧ (*concl* = subst-axiom (*exp*, *var*, *term*)))
→ proves (subst-axiom-proof (*exp*, *var*, *term*), *concl*)

THEOREM: equal-axiom1-proves
(function (*fn*)
∧ var-list (*vars1*, degree (*fn*))
∧ var-list (*vars2*, degree (*fn*))
∧ formula (*concl*, **t**, 0)
∧ (*concl* = equal-axiom1 (*vars1*, *vars2*, *fn*)))
→ proves (equal-axiom1-proof (*fn*, *vars1*, *vars2*), *concl*)

THEOREM: equal-axiom2-proves
(predicate (*pr*)
∧ var-list (*vars1*, degree (*pr*))
∧ var-list (*vars2*, degree (*pr*))
∧ formula (*concl*, **t**, 0)
∧ (*concl* = equal-axiom2 (*vars1*, *vars2*, *pr*)))
→ proves (equal-axiom2-proof (*pr*, *vars1*, *vars2*), *concl*)

THEOREM: ident-axiom-proves
(variable $(var) \land (concl = \text{ident-axiom}(var)) \land \text{formula}(concl, \mathtt{t}, \mathtt{0}))$
$\rightarrow$ proves (ident-axiom-proof (var), $concl)$

THEOREM: expan-proof-proves
(formula $(a, \mathtt{t}, \mathtt{0}) \land$ proves $(pf, b) \land (concl = \text{f-or}(a, b)))$
$\rightarrow$ proves (expan-proof (a, b, pf), $concl)$

THEOREM: contrac-proof-proves
proves $(pf, \text{f-or}(a, a)) \rightarrow$ proves (contrac-proof (a, pf), $a)$

THEOREM: assoc-proof-proves
(proves $(pf, \text{f-or}(a, \text{f-or}(b, c))) \land (concl = \text{f-or}(\text{f-or}(a, b), c)))$
$\rightarrow$ proves (assoc-proof (a, b, c, pf), $concl)$

THEOREM: cut-proof-proves
(proves $(pf1, \text{f-or}(a, b))$
$\land$ proves $(pf2, \text{f-or}(\text{f-not}(a), c))$
$\land$ $(concl = \text{f-or}(b, c)))$
$\rightarrow$ proves (cut-proof $(a, b, c, pf1, pf2)$, $concl)$

```
;disabling the proof-constructors since the lemmas above show they work.
```

EVENT: Enable prop-axiom-proof; name this event ‘g2752’.

EVENT: Enable subst-axiom-proof; name this event ‘g2753’.

EVENT: Enable equal-axiom1-proof; name this event ‘g2754’.

EVENT: Enable equal-axiom2-proof; name this event ‘g2755’.

EVENT: Enable ident-axiom-proof; name this event ‘g2756’.

EVENT: Enable expan-proof; name this event ‘g2759’.

EVENT: Enable contrac-proof; name this event ‘g2760’.

EVENT: Enable assoc-proof; name this event ‘g2761’.

EVENT: Enable cut-proof; name this event ‘g2762’.

THEOREM: forsome-intro-proves
(proves (pf, f-implies (a, b))
 ∧ ($var \notin$ collect-free (b, **t**))
 ∧ variable (var)
 ∧ ($a\text{-}prime$ = f-implies (forsome (var, a), b)))
→ proves (forsome-intro-proof (var, a, b, pf), $a\text{-}prime$)

EVENT: Enable forsome-intro-proof; name this event 'g2763'.

EVENT: Enable prf; name this event 'g2764'.

EVENT: Enable proves; name this event 'g2765'.

DEFINITION:
commut-proof (a, b, pf) = cut-proof (a, b, a, pf, prop-axiom-proof (a))

`;The first derived inference rule - commutativity of disjunction.`

THEOREM: commut-proof-proves
(proves (pf, f-or (a, b)) ∧ formula (f-or (a, b), **t**, `0`) ∧ ($concl$ = f-or (b, a)))
→ proves (commut-proof (a, b, pf), $concl$)

EVENT: Enable commut-proof; name this event 'g2766'.

`;Modus Ponens.`

DEFINITION:
detach-proof (a, b, $pf1$, $pf2$)
= contrac-proof (b,
    cut-proof (a,
        b,
        b,
        commut-proof (b, a, expan-proof (b, a, $pf1$)),
        $pf2$))

THEOREM: detach-proof-proves1
(proves ($pf1$, a) ∧ proves ($pf2$, f-implies (a, b)) ∧ formula (b, **t**, `0`))
→ proves (detach-proof (a, b, $pf1$, $pf2$), b)

EVENT: Enable detach-proof; name this event 'g2767'.

DEFINITION:
proves-list ($pflist$, $explist$)

$=$ **if** *explist* $\simeq$ **nil** **then** *pflist* $=$ **nil**
else proves (car (*pflist*), car (*explist*))
$\land$ proves-list (cdr (*pflist*), cdr (*explist*)) **endif**

DEFINITION:
list-implies (*list*, *conc*)
$=$ **if** *list* $\simeq$ **nil** **then** *conc*
elseif cdr (*list*) $\simeq$ **nil** **then** f-implies (car (*list*), *conc*)
else f-implies (car (*list*), list-implies (cdr (*list*), *conc*)) **endif**

DEFINITION:
list-detach-proof (*alist*, *b*, *pflist*, *pf2*)
$=$ **if** *alist* $\simeq$ **nil** **then** *pf2*
elseif cdr (*alist*) $\simeq$ **nil** **then** detach-proof (car (*alist*), *b*, car (*pflist*), *pf2*)
else list-detach-proof (cdr (*alist*),
b,
cdr (*pflist*),
detach-proof (car (*alist*),
list-implies (cdr (*alist*), *b*),
car (*pflist*),
pf2)) **endif**

`;Chained Modus Ponens.`

THEOREM: detach-list-implies
(list (*c*)
$\land$ proves (*pf*, *a*)
$\land$ proves (*pf2*, list-implies (cons (*a*, *c*), *b*))
$\land$ formula (*a*, **t**, 0)
$\land$ formula (list-implies (*c*, *b*), **t**, 0))
$\rightarrow$ proves (detach-proof (*a*, list-implies (*c*, *b*), *pf*, *pf2*), list-implies (*c*, *b*))

THEOREM: formula-list-implies
(formula (list-implies (*alist*, *b*), **t**, 0) $\land$ listp (*alist*))
$\rightarrow$ formula (list-implies (cdr (*alist*), *b*), **t**, 0)

THEOREM: detach-rule-corr
(proves-list (*pflist*, *alist*)
$\land$ proves (*pf2*, list-implies (*alist*, *b*))
$\land$ formula (*b*, **t**, 0))
$\rightarrow$ proves (list-detach-proof (*alist*, *b*, *pflist*, *pf2*), *b*)

EVENT: Enable list-detach-proof; name this event 'g0220'.

EVENT: Enable detach-list-implies; name this event 'g0221'.

DEFINITION:
rt-expan-proof $(a,\ b,\ pf)$ = commut-proof $(b,\ a,$ expan-proof $(b,\ a,\ pf))$

THEOREM: rt-expan-proof-proves
(proves $(pf,\ a)$ $\wedge$ formula $(b,\ \mathbf{t},\ 0)$ $\wedge$ $(concl =$ f-or $(a,\ b)))$
$\rightarrow$ proves (rt-expan-proof $(a,\ b,\ pf),\ concl)$

EVENT: Enable rt-expan-proof; name this event 'g0227'.

```
;Takes list of formulas and returns disjunction.
```

DEFINITION:
make-disjunct $(flist)$
= **if** $flist \simeq$ **nil** **then** **nil**
elseif cdr $(flist) \simeq$ **nil** **then** car $(flist)$
else f-or (car $(flist)$, make-disjunct (cdr $(flist)$)) **endif**

DEFINITION:
m1-proof $(exp,\ flist,\ pf)$
= **if** $flist \simeq$ **nil** **then** **nil**
elseif cdr $(flist) \simeq$ **nil** **then** pf
elseif $exp =$ car $(flist)$
then rt-expan-proof (car $(flist)$, make-disjunct (cdr $(flist)$), pf)
else expan-proof (car $(flist)$,
make-disjunct (cdr $(flist)$),
m1-proof $(exp$, cdr $(flist)$, pf)) **endif**

THEOREM: m1-proof-proves1
(formula (make-disjunct $(flist)$, $\mathbf{t}$, 0) $\wedge$ $(exp \in flist)$ $\wedge$ proves $(pf,\ exp))$
$\rightarrow$ proves (m1-proof $(exp,\ flist,\ pf)$, make-disjunct $(flist)$)

EVENT: Enable m1-proof; name this event 'g0228'.

DEFINITION:
rt-assoc-proof $(a,\ b,\ c,\ pf)$
= commut-proof (f-or $(b,\ c)$,
a,
assoc-proof $(b$,
c,
a,
commut-proof (f-or $(c,\ a)$,
b,
assoc-proof $(c$,
a,

b,
commut-proof (f-or (a,
b),
c,
pf)))))

THEOREM: rt-assoc-proof-proves
(proves (pf, f-or (f-or (a, b), c))
∧ formula (a, **t**, 0)
∧ formula (b, **t**, 0)
∧ formula (c, **t**, 0)
∧ ($concl$ = f-or (a, f-or (b, c))))
→ proves (rt-assoc-proof (a, b, c, pf), $concl$)

EVENT: Enable rt-assoc-proof; name this event ‘g0231’.

DEFINITION:
insert-proof (a, b, c, pf)
= commut-proof (f-or (a, c),
b,
assoc-proof (a,
c,
b,
expan-proof (a, f-or (c, b), commut-proof (b, c, pf))))

THEOREM: insert-proof-proves
(proves (pf, f-or (b, c))
∧ formula (a, **t**, 0)
∧ formula (b, **t**, 0)
∧ formula (c, **t**, 0)
∧ ($concl$ = f-or (b, f-or (a, c))))
→ proves (insert-proof (a, b, c, pf), $concl$)

EVENT: Enable insert-proof; name this event ‘g0232’.

DEFINITION:
m2-proof-step ($exp1$, $exp2$, $flist$, pf)
= **if** $flist$ ≃ **nil** **then nil**
elseif cdr ($flist$) ≃ **nil**
then if $exp2$ = car ($flist$) **then** pf
else nil endif
elseif $exp2$ = car ($flist$)
then rt-assoc-proof ($exp1$,
$exp2$,

make-disjunct (cdr (*flist*)),
rt-expan-proof (f-or (*exp1*, *exp2*),
make-disjunct (cdr (*flist*)),
pf))
else insert-proof (car (*flist*),
exp1,
make-disjunct (cdr (*flist*)),
m2-proof-step (*exp1*, *exp2*, cdr (*flist*), *pf*)) **endif**

THEOREM: m2-proof-step-proves
(formula (make-disjunct (*flist*), **t**, 0)
∧ (*exp2* ∈ *flist*)
∧ formula (*exp1*, **t**, 0)
∧ formula (*exp2*, **t**, 0)
∧ proves (*pf*, f-or (*exp1*, *exp2*)))
→ proves (m2-proof-step (*exp1*, *exp2*, *flist*, *pf*),
f-or (*exp1*, make-disjunct (*flist*)))

THEOREM: m2-proof-step-proves1
(formula (make-disjunct (*flist*), **t**, 0)
∧ (*exp2* ∈ *flist*)
∧ formula (*exp1*, **t**, 0)
∧ formula (*exp2*, **t**, 0)
∧ proves (*pf*, f-or (*exp1*, *exp2*))
∧ (*concl* = f-or (*exp1*, make-disjunct (*flist*))))
→ proves (m2-proof-step (*exp1*, *exp2*, *flist*, *pf*), *concl*)

EVENT: Enable m2-proof-step; name this event ‘g0233’.

EVENT: Enable m2-proof-step-proves; name this event ‘g0234’.

DEFINITION:
m2-proof (*exp1*, *exp2*, *flist*, *pf*)
= **if** *flist* ≃ **nil then nil**
elseif *exp1* = *exp2* **then** m1-proof (*exp1*, *flist*, contrac-proof (*exp1*, *pf*))
elseif *exp1* = car (*flist*)
then m2-proof-step (*exp1*, *exp2*, cdr (*flist*), *pf*)
elseif *exp2* = car (*flist*)
then m2-proof-step (*exp2*, *exp1*, cdr (*flist*), commut-proof (*exp1*, *exp2*, *pf*))
else expan-proof (car (*flist*),
make-disjunct (cdr (*flist*)),
m2-proof (*exp1*, *exp2*, cdr (*flist*), *pf*)) **endif**

THEOREM: m1-proof-proves

(formula (make-disjunct (*flist*), **t**, 0)
∧ (*exp* ∈ *flist*)
∧ proves (*pf*, *exp*)
∧ (*concl* = make-disjunct (*flist*)))
→ proves (m1-proof (*exp*, *flist*, *pf*), *concl*)

THEOREM: m2-proof-proves
(formula (make-disjunct (*flist*), **t**, 0)
∧ formula (*exp1*, **t**, 0)
∧ formula (*exp2*, **t**, 0)
∧ (*exp1* ∈ *flist*)
∧ (*exp2* ∈ *flist*)
∧ proves (*pf*, f-or (*exp1*, *exp2*)))
→ proves (m2-proof (*exp1*, *exp2*, *flist*, *pf*), make-disjunct (*flist*))

THEOREM: m2-proof-proves1
(formula (make-disjunct (*flist*), **t**, 0)
∧ formula (*exp1*, **t**, 0)
∧ formula (*exp2*, **t**, 0)
∧ (*exp1* ∈ *flist*)
∧ (*exp2* ∈ *flist*)
∧ proves (*pf*, f-or (*exp1*, *exp2*))
∧ (*concl* = make-disjunct (*flist*)))
→ proves (m2-proof (*exp1*, *exp2*, *flist*, *pf*), *concl*)

EVENT: Enable m2-proof; name this event 'g0235'.

EVENT: Enable m2-proof-proves; name this event 'g0236'.

DEFINITION:
m3-proof (*exp1*, *exp2*, *flist2*, *pf*)
= contrac-proof (make-disjunct (*flist2*),
  contrac-proof (f-or (make-disjunct (*flist2*),
   make-disjunct (*flist2*)),
   m2-proof (f-or (make-disjunct (*flist2*),
    make-disjunct (*flist2*)),
    exp1,
    cons (f-or (make-disjunct (*flist2*),
     make-disjunct (*flist2*)),
     cons (make-disjunct (*flist2*),
      flist2)),
    assoc-proof (make-disjunct (*flist2*),
     make-disjunct (*flist2*),
     exp1,

```
commut-proof (f-or (make-disjunct (flist2),
                    exp1),
               make-disjunct (flist2),
               m2-proof (f-or (make-disjunct (flist2),
                               exp1),
                         exp2,
                         cons (f-or (make-disjunct (fli
                                     exp1),
                               flist2),
                         assoc-proof (make-disjunct (
                                      exp1,
                                      exp2,
                                      commut-proof (
```

DEFINITION:
m-proof (*flist1*, *flist2*, *pf*)

```
=   if flist1 ≃ nil then nil
    elseif cdr (flist1) ≃ nil then m1-proof (car (flist1), flist2, pf)
    elseif cddr (flist1) ≃ nil
    then m2-proof (car (flist1), cadr (flist1), flist2, pf)
    else m3-proof (car (flist1),
                   cadr (flist1),
                   flist2,
                   m-proof (cons (f-or (car (flist1), cadr (flist1)),
                                  cddr (flist1)),
                            cons (f-or (car (flist1), cadr (flist1)),
                                  flist2),
                            assoc-proof (car (flist1),
                                         cadr (flist1),
                                         make-disjunct (cddr (flist1)),
                                         pf))) endif
```

THEOREM: subset-cons
subset $(x, y) \rightarrow$ subset $(x, \text{cons}\,(z, y))$

DEFINITION:
form-list (*flist*)

```
=   if listp (flist)
    then formula (car (flist), t, 0) ∧ form-list (cdr (flist))
    else t endif
```

THEOREM: formlist-formula-make-disj
(form-list (*flist*) $\land$ listp (*flist*)) $\rightarrow$ formula (make-disjunct (*flist*), **t**, 0)

THEOREM: m3-proof-proves
(formula (*exp1*, **t**, 0)
 ∧ formula (*exp2*, **t**, 0)
 ∧ form-list (*flist2*)
 ∧ proves (*pf*, make-disjunct (cons (f-or (*exp1*, *exp2*), *flist2*)))
 ∧ (*exp1* ∈ *flist2*)
 ∧ (*exp2* ∈ *flist2*))
→ proves (m3-proof (*exp1*, *exp2*, *flist2*, *pf*), make-disjunct (*flist2*))

EVENT: Enable m3-proof; name this event 'g0222'.

THEOREM: m3-proof-proves1
(formula (*exp1*, **t**, 0)
 ∧ formula (*exp2*, **t**, 0)
 ∧ form-list (*flist2*)
 ∧ proves (*pf*, make-disjunct (cons (f-or (*exp1*, *exp2*), *flist2*)))
 ∧ (*exp1* ∈ *flist2*)
 ∧ (*exp2* ∈ *flist2*)
 ∧ (*concl* = make-disjunct (*flist2*)))
→ proves (m3-proof (*exp1*, *exp2*, *flist2*, *pf*), *concl*)

EVENT: Enable m3-proof-proves; name this event 'g0229'.

```
;The subset lemma
```

THEOREM: m-proof-proves
(form-list (*flist1*)
 ∧ listp (*flist1*)
 ∧ form-list (*flist2*)
 ∧ listp (*flist2*)
 ∧ subset (*flist1*, *flist2*)
 ∧ proves (*pf*, make-disjunct (*flist1*)))
→ proves (m-proof (*flist1*, *flist2*, *pf*), make-disjunct (*flist2*))

EVENT: Enable m-proof; name this event 'g0247'.

THEOREM: m-proof-proves1
(form-list (*flist1*)
 ∧ form-list (*flist2*)
 ∧ subset (*flist1*, *flist2*)
 ∧ proves (*pf*, make-disjunct (*flist1*))
 ∧ (*concl* = make-disjunct (*flist2*)))
→ proves (m-proof (*flist1*, *flist2*, *pf*), *concl*)

```
;Disjunctions.
```

DEFINITION: or-type (*exp*) = (*exp* = f-or (cadr (*exp*), caddr (*exp*)))

```
;Negation of disjunctions.
```

DEFINITION:
nor-type (*exp*) = (*exp* = f-not (f-or (cadadr (*exp*), caddadr (*exp*))))

EVENT: Enable atomp; name this event 'g0250'.

```
;Elementary and negation of elementary formulas
```

DEFINITION:
elem-form (*exp*) = (atomp (*exp*) $\lor$ (*exp* = forsome (cadr (*exp*), caddr (*exp*))))

DEFINITION:
neg-elem-form (*exp*)
= ((*exp* = f-not (cadr (*exp*))) $\land$ elem-form (arg1 (*exp*)))

DEFINITION:
prop-atomp (*exp*) = (elem-form (*exp*) $\lor$ neg-elem-form (*exp*))

```
;Double-negations
```

DEFINITION:
dble-neg-type (*exp*) = (*exp* = f-not (f-not (cadadr (*exp*))))

EVENT: Disable atomp; name this event 'g0251'.

THEOREM: dble-neg-not-prop-atomp
dble-neg-type (*exp*) $\rightarrow$ ($\neg$ prop-atomp (*exp*))

THEOREM: or-type-not-prop-atomp
or-type (*exp*) $\rightarrow$ ($\neg$ prop-atomp (*exp*))

THEOREM: nor-type-not-prop-atomp
nor-type (*exp*) $\rightarrow$ ($\neg$ prop-atomp (*exp*))

DEFINITION:
list-count (*list*)
= **if** *list* $\simeq$ **nil** **then** 0
else (1 + count (car (*list*))) + list-count (cdr (*list*)) **endif**

Definition:
neg-list (*exp*, *list*)
= **if** *list* ≃ **nil** **then f**
else neg (*exp*, car (*list*)) ∨ neg-list (*exp*, cdr (*list*)) **endif**

Theorem: lessp-list-count
listp (*flist*) → (list-count (cdr (*flist*)) < list-count (*flist*))

Theorem: or-type-list-count
(or-type (car (*flist*)) ∧ listp (*flist*))
→ (list-count (cons (cadar (*flist*), cons (caddar (*flist*), cdr (*flist*))))
< list-count (*flist*))

Theorem: nor-type-list-count1
(listp (*flist*) ∧ nor-type (car (*flist*)))
→ (list-count (cons (list (`'not`, cadadar (*flist*)), cdr (*flist*)))
< list-count (*flist*))

Theorem: nor-type-list-count2
(listp (*flist*) ∧ nor-type (car (*flist*)))
→ (list-count (cons (list (`'not`, caddadar (*flist*)), cdr (*flist*)))
< list-count (*flist*))

Theorem: dble-neg-list-count
(listp (*flist*) ∧ dble-neg-type (car (*flist*)))
→ (list-count (cons (cadadar (*flist*), cdr (*flist*))) < list-count (*flist*))

Event: Enable prop-atomp; name this event 'g0230'.

Event: Enable or-type; name this event 'g0237'.

Event: Enable nor-type; name this event 'g0238'.

Event: Enable dble-neg-type; name this event 'g0239'.

Event: Enable list-count; name this event 'g0240'.

```
;The tautology-checker.
```

Definition:
tautologyp1 (*flist*, *auxlist*)
= **if** *flist* ≃ **nil** **then f**
elseif prop-atomp (car (*flist*))

then neg-list (car (*flist*), *auxlist*)
 ∨ tautologyp1 (cdr (*flist*), cons (car (*flist*), *auxlist*))
elseif or-type (car (*flist*))
then tautologyp1 (cons (arg1 (car (*flist*)),
 cons (arg2 (car (*flist*)), cdr (*flist*))),
 auxlist)
elseif nor-type (car (*flist*))
then tautologyp1 (cons (f-not (arg1 (arg1 (car (*flist*)))), cdr (*flist*)),
 auxlist)
 ∧ tautologyp1 (cons (f-not (arg2 (arg1 (car (*flist*)))), cdr (*flist*)),
 auxlist)
elseif dble-neg-type (car (*flist*))
then tautologyp1 (cons (arg1 (arg1 (car (*flist*))), cdr (*flist*)), *auxlist*)
else f endif

THEOREM: form-list-append
(form-list (x) ∧ form-list (y)) → form-list (append (x, y))

DEFINITION:
neg-proof (*exp1*, *exp2*)
= **if** *exp1* = f-not (*exp2*) **then** prop-axiom-proof (*exp2*)
else commut-proof (*exp2*, *exp1*, prop-axiom-proof (*exp1*)) **endif**

THEOREM: neg-proof-proves
(formula (*exp1*, **t**, 0)
∧ formula (*exp2*, **t**, 0)
∧ neg (*exp1*, *exp2*)
∧ (*concl* = f-or (*exp1*, *exp2*)))
→ proves (neg-proof (*exp1*, *exp2*), *concl*)

EVENT: Enable neg-proof; name this event 'g0245'.

THEOREM: neg-list-reduc
neg-list (*exp*, *flist*)
= ((f-not (*exp*) ∈ *flist*)
 ∨ ((*exp* = f-not (cadr (*exp*))) ∧ (cadr (*exp*) ∈ *flist*)))

DEFINITION:
neg-list-proof (*exp*, *flist*)
= **if** f-not (*exp*) ∈ *flist*
then m2-proof (*exp*,
 f-not (*exp*),
 cons (*exp*, *flist*),
 neg-proof (*exp*, f-not (*exp*)))

else m2-proof (*exp*,
cadr (*exp*),
cons (*exp*, *flist*),
neg-proof (*exp*, cadr (*exp*))) **endif**

THEOREM: neg-list-proof-proves
(formula (*exp*, **t**, 0)
∧ form-list (*flist*)
∧ neg-list (*exp*, *flist*)
∧ (*concl* = make-disjunct (cons (*exp*, *flist*))))
→ proves (neg-list-proof (*exp*, *flist*), *concl*)

EVENT: Enable neg-list-proof; name this event 'g0256'.

THEOREM: subset-ident
subset (x, x)

THEOREM: subset-car
subset (x, cons (y, x))

THEOREM: subset-append
subset (cons (*exp*, *list2*), append (cons (*exp*, *list1*), *list2*))

THEOREM: nlistp-neg-list
(*list* ≃ **nil**) → (¬ neg-list (*exp*, *list*))

DEFINITION:
prop-atom-proof1 (*flist1*, *flist2*)
= m-proof (cons (car (*flist1*), *flist2*),
append (*flist1*, *flist2*),
neg-list-proof (car (*flist1*), *flist2*))

THEOREM: prop-atom-proof1-proves
(form-list (*flist1*)
∧ listp (*flist1*)
∧ form-list (*flist2*)
∧ neg-list (car (*flist1*), *flist2*)
∧ (*concl* = make-disjunct (append (*flist1*, *flist2*))))
→ proves (prop-atom-proof1 (*flist1*, *flist2*), *concl*)

EVENT: Enable prop-atom-proof1; name this event 'g0259'.

THEOREM: subset-append-car
subset (append (*list1*, cons (*exp*, *list2*)), append (cons (*exp*, *list1*), *list2*))

THEOREM: form-list-append-car
(form-list (cons (*exp*, *list1*)) ∧ form-list (*list2*))
→ form-list (append (*list1*, cons (*exp*, *list2*)))

DEFINITION:
prop-atom-proof2 (*flist1*, *flist2*, *pf*)
= m-proof (append (cdr (*flist1*), cons (car (*flist1*), *flist2*)),
append (*flist1*, *flist2*),
pf)

THEOREM: prop-atom-proof2-proves
(form-list (*flist1*)
∧ listp (*flist1*)
∧ form-list (*flist2*)
∧ proves (*pf*,
make-disjunct (append (cdr (*flist1*), cons (car (*flist1*), *flist2*))))
∧ (*concl* = make-disjunct (append (*flist1*, *flist2*))))
→ proves (prop-atom-proof2 (*flist1*, *flist2*, *pf*), *concl*)

EVENT: Enable prop-atom-proof2; name this event 'g0258'.

DEFINITION:
cancel-proof (a, b, *pf1*, *pf2*)
= contrac-proof (b, cut-proof (a, b, b, *pf2*, rt-expan-proof (f-not (a), b, *pf1*)))

THEOREM: cancel-proof-proves
(proves (*pf1*, f-not (a))
∧ proves (*pf2*, f-or (a, b))
∧ formula (a, **t**, 0)
∧ formula (b, **t**, 0))
→ proves (cancel-proof (a, b, *pf1*, *pf2*), b)

EVENT: Enable cancel-proof; name this event 'g0255'.

DEFINITION:
nlistp-nor-type-proof (a, b, *pf1*, *pf2*)
= cancel-proof (b,
f-not (f-or (a, b)),
pf2,
cancel-proof (a,
f-or (b, f-not (f-or (a, b))),
pf1,
m-proof (list (f-not (f-or (a, b)), a, b),
list (a, b, f-not (f-or (a, b))),
prop-axiom-proof (f-or (a, b)))))

THEOREM: nlistp-nor-type-proof-proves

```
 (formula (a, t, 0)
  ∧   formula (b, t, 0)
  ∧   proves (pf1, f-not (a))
  ∧   proves (pf2, f-not (b))
  ∧   (concl = f-not (f-or (a, b))))
 →   proves (nlistp-nor-type-proof (a, b, pf1, pf2), concl)
```

DEFINITION:

```
listp-nor-type-proof (a, b, c, pf1, pf2)
=   m-proof (list (f-not (f-or (a, b)), c, c),
             list (f-not (f-or (a, b)), c),
             rt-assoc-proof (f-not (f-or (a, b)),
                             c,
                             c,
                             cut-proof (b,
                                        f-or (f-not (f-or (a, b)), c),
                                        c,
                                        rt-assoc-proof (b,
                                                        f-not (f-or (a, b)),
                                                        c,
                                                        cut-proof (a,
                                                                   f-or (b,
                                                                         f-not (f-or (a,
                                                                                      b))),
                                                                   c,
                                                                   m-proof (list (f-not (f-or (a,
                                                                                               b)),
                                                                                  a,
                                                                                  b),
                                                                            list (a,
                                                                                  b,
                                                                                  f-not (f-or (a,
                                                                                               b))),
                                                                            prop-axiom-proof (f-or (a,
                                                                                                    b))),
                                                                   pf1)),
                                        pf2)))
```

THEOREM: listp-nor-type-proof-proves

```
 (formula (a, t, 0)
  ∧   formula (b, t, 0)
  ∧   formula (c, t, 0)
  ∧   proves (pf1, f-or (f-not (a), c))
```

$\wedge$ proves (*pf2*, f-or (f-not (b), c))
$\wedge$ (*concl* = f-or (f-not (f-or (a, b)), c)))
$\rightarrow$ proves (listp-nor-type-proof (a, b, c, *pf1*, *pf2*), *concl*)

EVENT: Enable m-proof-proves; name this event ‘g0242’.

EVENT: Enable nlistp-nor-type-proof; name this event ‘g0243’.

EVENT: Enable listp-nor-type-proof; name this event ‘g0244’.

DEFINITION:
nor-type-proof (a, b, *clist*, *pf1*, *pf2*)
= **if** *clist* $\simeq$ **nil then** nlistp-nor-type-proof (a, b, *pf1*, *pf2*)
else listp-nor-type-proof (a, b, make-disjunct (*clist*), *pf1*, *pf2*) **endif**

EVENT: Disable nor-type; name this event ‘g0292’.

THEOREM: nor-type-proof-proves
(nor-type (*exp*)
$\wedge$ formula (*exp*, **t**, 0)
$\wedge$ form-list (*clist*)
$\wedge$ proves (*pf1*, make-disjunct (cons (f-not (cadadr (*exp*)), *clist*)))
$\wedge$ proves (*pf2*, make-disjunct (cons (f-not (caddadr (*exp*)), *clist*)))
$\wedge$ (*concl* = make-disjunct (cons (*exp*, *clist*))))
$\rightarrow$ proves (nor-type-proof (cadadr (*exp*), caddadr (*exp*), *clist*, *pf1*, *pf2*), *concl*)

DEFINITION:
nlistp-dble-neg-proof (a, *pf*)
= contrac-proof (f-not (f-not (a)),
cut-proof (a,
f-not (f-not (a)),
f-not (f-not (a)),
rt-expan-proof (a, f-not (f-not (a)), *pf*),
commut-proof (f-not (f-not (a)),
f-not (a),
prop-axiom-proof (f-not (a)))))

EVENT: Enable nor-type-proof; name this event ‘g0248’.

THEOREM: nlistp-dble-neg-proof-proves
(formula (a, **t**, 0) $\wedge$ proves (*pf*, a) $\wedge$ (*concl* = f-not (f-not (a))))
$\rightarrow$ proves (nlistp-dble-neg-proof (a, *pf*), *concl*)

Event: Enable nlistp-dble-neg-proof; name this event 'g0249'.

Definition:
listp-dble-neg-proof (a, c, pf)
= commut-proof (c,
f-not (f-not (a)),
cut-proof (a,
c,
f-not (f-not (a)),
pf,
commut-proof (f-not (f-not (a)),
f-not (a),
prop-axiom-proof (f-not (a)))))

Theorem: listp-dble-neg-proof-proves
(formula (a, **t**, 0)
∧ formula (c, **t**, 0)
∧ proves (pf, f-or (a, c))
∧ ($concl$ = f-or (f-not (f-not (a)), c)))
→ proves (listp-dble-neg-proof (a, c, pf), $concl$)

Event: Enable listp-dble-neg-proof; name this event 'g0203'.

Definition:
dble-neg-type-proof (a, $clist$, pf)
= **if** $clist \simeq$ **nil** **then** nlistp-dble-neg-proof (a, pf)
else listp-dble-neg-proof (a, make-disjunct ($clist$), pf) **endif**

Event: Disable dble-neg-type; name this event 'g0252'.

Theorem: dble-neg-type-proof-proves
(dble-neg-type (exp)
∧ formula (exp, **t**, 0)
∧ form-list ($clist$)
∧ proves (pf, make-disjunct (cons (cadadr (exp), $clist$)))
∧ ($concl$ = make-disjunct (cons (exp, $clist$))))
→ proves (dble-neg-type-proof (cadadr (exp), $clist$, pf), $concl$)

Definition:
or-type-proof (a, b, $clist$, pf)
= **if** $clist \simeq$ **nil** **then** pf
else assoc-proof (a, b, make-disjunct ($clist$), pf) **endif**

Event: Disable or-type; name this event 'g0260'.

THEOREM: or-type-proof-proves
(or-type (car (*flist1*))
 ∧ form-list (*flist1*)
 ∧ listp (*flist1*)
 ∧ form-list (*flist2*)
 ∧ proves (*pf*,
  make-disjunct (append (cons (cadar (*flist1*),
   cons (caddar (*flist1*), cdr (*flist1*))),
   flist2)))
 ∧ (*concl* = make-disjunct (append (*flist1*, *flist2*))))
→ proves (or-type-proof (cadar (*flist1*),
  caddar (*flist1*),
  append (cdr (*flist1*), *flist2*),
  pf),
  concl)

EVENT: Enable or-type-proof; name this event 'g0271'.

THEOREM: or-type-form-list
(or-type (car (*flist*)) ∧ form-list (*flist*) ∧ listp (*flist*))
→ form-list (cons (cadar (*flist*), cons (caddar (*flist*), cdr (*flist*))))

THEOREM: nor-type-form-list
(nor-type (car (*flist*)) ∧ form-list (*flist*) ∧ listp (*flist*))
→ form-list (cons (list (`'not`, cadadar (*flist*)), cdr (*flist*)))

THEOREM: nor-type-form-list2
(nor-type (car (*flist*)) ∧ form-list (*flist*) ∧ listp (*flist*))
→ form-list (cons (list (`'not`, caddadar (*flist*)), cdr (*flist*)))

THEOREM: dble-neg-type-form-list
(dble-neg-type (car (*flist*)) ∧ form-list (*flist*) ∧ listp (*flist*))
→ form-list (cons (cadadar (*flist*), cdr (*flist*)))

EVENT: Enable or-type; name this event 'g0272'.

EVENT: Enable nor-type; name this event 'g0273'.

EVENT: Enable dble-neg-type; name this event 'g0274'.

EVENT: Enable dble-neg-type-proof; name this event 'g0254'.

```
;The proof-constructor for tautologies.
```

Definition:

```
taut-proof1 (flist, auxlist)
=   if flist ≃ nil then nil
    elseif prop-atomp (car (flist))
    then if neg-list (car (flist), auxlist)
           then prop-atom-proof1 (flist, auxlist)
           else prop-atom-proof2 (flist,
                                  auxlist,
                                  taut-proof1 (cdr (flist),
                                               cons (car (flist), auxlist))) endif
    elseif or-type (car (flist))
    then or-type-proof (arg1 (car (flist)),
                        arg2 (car (flist)),
                        append (cdr (flist), auxlist),
                        taut-proof1 (cons (arg1 (car (flist)),
                                           cons (arg2 (car (flist)), cdr (flist))),
                                     auxlist))
    elseif nor-type (car (flist))
    then nor-type-proof (arg1 (arg1 (car (flist))),
                         arg2 (arg1 (car (flist))),
                         append (cdr (flist), auxlist),
                         taut-proof1 (cons (f-not (arg1 (arg1 (car (flist)))),
                                            cdr (flist)),
                                      auxlist),
                         taut-proof1 (cons (f-not (arg2 (arg1 (car (flist)))),
                                            cdr (flist)),
                                      auxlist))
    elseif dble-neg-type (car (flist))
    then dble-neg-type-proof (arg1 (arg1 (car (flist))),
                              append (cdr (flist), auxlist),
                              taut-proof1 (cons (arg1 (arg1 (car (flist))),
                                                 cdr (flist)),
                                           auxlist))
    else nil endif
```

```
;Every tautology has a proof (when AUXLIST is NIL)
```

Theorem: taut-thm1

```
 (form-list (flist) ∧ form-list (auxlist) ∧ tautologyp1 (flist, auxlist))
 →  proves (taut-proof1 (flist, auxlist),
            make-disjunct (append (flist, auxlist)))
```

Event: Enable taut-proof1; name this event 'g0275'.

Theorem: taut-thm2

$$
\begin{array}{l}
(\text{form-list}\,(\mathit{flist}) \\
\;\wedge\;\; \text{form-list}\,(\mathit{auxlist}) \\
\;\wedge\;\; \text{tautologyp1}\,(\mathit{flist},\ \mathit{auxlist}) \\
\;\wedge\;\; (\mathit{concl} = \text{make-disjunct}\,(\text{append}\,(\mathit{flist},\ \mathit{auxlist})))) \\
\rightarrow\;\; \text{proves}\,(\text{taut-proof1}\,(\mathit{flist},\ \mathit{auxlist}),\ \mathit{concl})
\end{array}
$$

THEOREM: listp-elem-form
$(\mathit{exp} \simeq \textbf{nil}) \rightarrow (\neg\ \text{elem-form}\,(\mathit{exp}))$

```
;Truth value evaluator.
```

DEFINITION:
$\text{eval}\,(\mathit{exp},\ \mathit{alist})$

$$
\begin{array}{ll}
= & \textbf{if } \mathit{exp} \simeq \textbf{nil} \textbf{ then } \textbf{f} \\
& \textbf{elseif } \text{elem-form}\,(\mathit{exp}) \textbf{ then } \mathit{exp} \in \mathit{alist} \\
& \textbf{elseif } \text{car}\,(\mathit{exp}) = \texttt{'not} \textbf{ then } \neg\ \text{eval}\,(\text{cadr}\,(\mathit{exp}),\ \mathit{alist}) \\
& \textbf{elseif } \text{car}\,(\mathit{exp}) = \texttt{'or} \\
& \textbf{then } \text{eval}\,(\text{cadr}\,(\mathit{exp}),\ \mathit{alist}) \vee \text{eval}\,(\text{caddr}\,(\mathit{exp}),\ \mathit{alist}) \\
& \textbf{else f endif}
\end{array}
$$

THEOREM: elem-form-eval
$\text{elem-form}\,(\mathit{exp}) \rightarrow (\text{eval}\,(\mathit{exp},\ \mathit{alist}) = (\mathit{exp} \in \mathit{alist}))$

THEOREM: nlistp-eval
$(\mathit{exp} \simeq \textbf{nil}) \rightarrow (\neg\ \text{eval}\,(\mathit{exp},\ \mathit{alist}))$

THEOREM: not-eval

$$
\begin{array}{l}
(\text{listp}\,(\mathit{exp}) \wedge (\text{car}\,(\mathit{exp}) = \texttt{'not})) \\
\rightarrow\;\; (\text{eval}\,(\mathit{exp},\ \mathit{alist}) = (\neg\ \text{eval}\,(\text{cadr}\,(\mathit{exp}),\ \mathit{alist})))
\end{array}
$$

THEOREM: or-eval

$$
\begin{array}{l}
(\text{listp}\,(\mathit{exp}) \wedge (\text{car}\,(\mathit{exp}) = \texttt{'or})) \\
\rightarrow\;\; (\text{eval}\,(\mathit{exp},\ \mathit{alist}) \\
\qquad = \;\; (\text{eval}\,(\text{cadr}\,(\mathit{exp}),\ \mathit{alist}) \vee \text{eval}\,(\text{caddr}\,(\mathit{exp}),\ \mathit{alist})))
\end{array}
$$

THEOREM: member-eval
$((\mathit{exp} \in \mathit{flist}) \wedge \text{eval}\,(\mathit{exp},\ \mathit{alist})) \rightarrow \text{eval}\,(\text{make-disjunct}\,(\mathit{flist}),\ \mathit{alist})$

THEOREM: eval-elem-form

$$
\begin{array}{l}
(\text{elem-form}\,(\mathit{exp}) \\
\;\wedge\;\; (\mathit{exp} \in \mathit{list}) \\
\;\wedge\;\; (\mathit{exp} \in \mathit{alist}) \\
\;\wedge\;\; (\mathit{concl} = \text{make-disjunct}\,(\mathit{list}))) \\
\rightarrow\;\; \text{eval}\,(\mathit{concl},\ \mathit{alist})
\end{array}
$$

Event: Disable or-type; name this event ‘g0278’.

Event: Disable nor-type; name this event ‘g0279’.

Event: Disable dble-neg-type; name this event ‘g0280’.

Event: Disable prop-atomp; name this event ‘g0281’.

Theorem: member-append
$(exp \in \text{append}(flist1, flist2)) = ((exp \in flist1) \vee (exp \in flist2))$

Theorem: eval-neg-elem-form
$((exp \in list)$
$\wedge \quad (\text{f-not}(exp) \in list)$
$\wedge \quad (concl = \text{make-disjunct}(list)))$
$\rightarrow \quad \text{eval}(concl, alist)$

Theorem: eval-make-disjunct
$\text{eval}(\text{make-disjunct}(\text{append}(list1, list2)), alist)$
$= \quad (\text{eval}(\text{make-disjunct}(list1), alist)$
$\vee \quad \text{eval}(\text{make-disjunct}(list2), alist))$

Theorem: neg-list-eval
$(\text{listp}(flist1)$
$\wedge \quad \text{neg-list}(\text{car}(flist1), flist2)$
$\wedge \quad (concl = \text{make-disjunct}(\text{append}(flist1, flist2))))$
$\rightarrow \quad \text{eval}(concl, alist)$

Theorem: eval-prop-atomp
$(\text{listp}(flist1)$
$\wedge \quad \text{eval}(\text{make-disjunct}(\text{append}(\text{cdr}(flist1), \text{cons}(\text{car}(flist1), flist2))),$
$alist))$
$\rightarrow \quad \text{eval}(\text{make-disjunct}(\text{append}(flist1, flist2)), alist)$

Event: Enable eval; name this event ‘g1253’.

Theorem: eval-or-type
$(\text{listp}(flist1) \wedge \text{or-type}(\text{car}(flist1)))$
$\rightarrow \quad (\text{eval}(\text{make-disjunct}(\text{append}(flist1, flist2)), alist)$
$= \quad \text{eval}(\text{make-disjunct}(\text{append}(\text{cons}(\text{cadar}(flist1),$
$\text{cons}(\text{caddar}(flist1),$
$\text{cdr}(flist1))),$
$flist2)),$
$alist))$

THEOREM: eval-nor-type

```
 (listp (flist1) ∧ nor-type (car (flist1)))
 →   (eval (make-disjunct (append (flist1, flist2)), alist)
        =   (eval (make-disjunct (append (cons (f-not (cadadar (flist1)),
                                                     cdr (flist1)),
                                               flist2)),
                    alist)
              ∧   eval (make-disjunct (append (cons (f-not (caddadar (flist1)),
                                                        cdr (flist1)),
                                                  flist2)),
                        alist)))
```

THEOREM: eval-dble-neg-type

```
 (listp (flist1) ∧ dble-neg-type (car (flist1)))
 →   (eval (make-disjunct (append (flist1, flist2)), alist)
        =   eval (make-disjunct (append (cons (cadadar (flist1), cdr (flist1)),
                                              flist2)),
                   alist))
```

```
;All tautologies are logically-true.
```

THEOREM: taut-eval

```
 tautologyp1 (flist, auxlist)
 →   eval (make-disjunct (append (flist, auxlist)), alist)
```

THEOREM: not-eval-prop-atomp

```
 (listp (flist1)
  ∧   (¬ eval (make-disjunct (append (cdr (flist1), cons (car (flist1), flist2))),
                alist)))
 →   (¬ eval (make-disjunct (append (flist1, flist2)), alist))
```

THEOREM: prop-atomp-reduc

```
 prop-atomp (exp)
 =   (elem-form (exp)
       ∨   ((exp = f-not (cadr (exp))) ∧ elem-form (cadr (exp))))
```

EVENT: Enable elem-form; name this event ‘g0263’.

EVENT: Enable prop-atomp; name this event ‘g0264’.

DEFINITION:

```
prop-atomp-list (list)
=   if list ≃ nil then t
    else prop-atomp (car (list)) ∧ prop-atomp-list (cdr (list)) endif
```

DEFINITION:
falsify (*list*)
= **if** *list* $\simeq$ **nil** **then nil**
elseif car (*list*) = f-not (cadar (*list*))
then cons (cadar (*list*), falsify (cdr (*list*)))
else falsify (cdr (*list*)) **endif**

THEOREM: falsify-step
(f-not (*exp*) $\notin$ *auxlist*) $\rightarrow$ (*exp* $\notin$ falsify (*auxlist*))

THEOREM: prop-atomp-auxlist
(prop-atomp (*exp*)
$\land$ ($\neg$ neg-list (*exp*, *auxlist*))
$\land$ prop-atomp-list (*auxlist*))
$\rightarrow$ ($\neg$ eval (*exp*, falsify (cons (*exp*, *auxlist*))))

THEOREM: prop-atomp-auxlist2
(($\neg$ neg-list (*exp*, *auxlist*))
$\land$ prop-atomp-list (*auxlist*)
$\land$ prop-atomp (*exp*)
$\land$ ($\neg$ eval (make-disjunct (*auxlist*), falsify (*alist*))))
$\rightarrow$ ($\neg$ eval (make-disjunct (*auxlist*), falsify (cons (*exp*, *alist*))))

THEOREM: prop-atomp-falsify
(prop-atomp (*exp*)
$\land$ ($\neg$ neg-list (*exp*, *auxlist*))
$\land$ prop-atomp-list (*auxlist*)
$\land$ ($\neg$ eval (make-disjunct (*auxlist*), falsify (*auxlist*))))
$\rightarrow$ ($\neg$ eval (make-disjunct (cons (*exp*, *auxlist*)), falsify (cons (*exp*, *auxlist*))))

EVENT: Disable prop-atomp; name this event ‘g0268’.

EVENT: Disable elem-form; name this event ‘g0269’.

EVENT: Enable atomp; name this event ‘g0204’.

THEOREM: formula-cases1
formula (*exp*, **t**, 0)
$\rightarrow$ (prop-atomp (*exp*)
$\lor$ or-type (*exp*)
$\lor$ nor-type (*exp*)
$\lor$ dble-neg-type (*exp*))

EVENT: Disable atomp; name this event 'g0205'.

EVENT: Enable prop-atomp; name this event 'g0296'.

EVENT: Enable or-type; name this event 'g0297'.

EVENT: Enable nor-type; name this event 'g0298'.

EVENT: Enable dble-neg-type; name this event 'g0299'.

THEOREM: formula-cases
((¬ dble-neg-type (*exp*))
 ∧ (¬ nor-type (*exp*))
 ∧ (¬ or-type (*exp*))
 ∧ (¬ prop-atomp (*exp*)))
→ (¬ formula (*exp*, **t**, `0`))

DEFINITION:
falsify-taut (*flist*, *auxlist*)
= **if** *flist* ≃ **nil** **then** falsify (*auxlist*)
  elseif prop-atomp (car (*flist*))
  then if neg-list (car (*flist*), *auxlist*) **then f**
    else falsify-taut (cdr (*flist*), cons (car (*flist*), *auxlist*)) **endif**
  elseif or-type (car (*flist*))
  then falsify-taut (cons (cadar (*flist*), cons (caddar (*flist*), cdr (*flist*))),
      auxlist)
  elseif nor-type (car (*flist*))
  then if falsify-taut (cons (f-not (caddadar (*flist*)), cdr (*flist*)), *auxlist*)
    then falsify-taut (cons (f-not (caddadar (*flist*)), cdr (*flist*)),
        auxlist)
    else falsify-taut (cons (f-not (cadadar (*flist*)), cdr (*flist*)),
        auxlist) **endif**
  elseif dble-neg-type (car (*flist*))
  then falsify-taut (cons (cadadar (*flist*), cdr (*flist*)), *auxlist*)
  else nil endif

THEOREM: append-nlistp
(*x* ≃ **nil**) → (append (*x*, *y*) = *y*)

THEOREM: not-falsify-taut
tautologyp1 (*flist*, *auxlist*) = (¬ falsify-taut (*flist*, *auxlist*))

`;Non-tautologies are falsifiable.`

Theorem: not-taut-false
(form-list (*flist*)
 ∧ prop-atomp-list (*auxlist*)
 ∧ (¬ eval (make-disjunct (*auxlist*), falsify (*auxlist*)))
 ∧ (¬ tautologyp1 (*flist*, *auxlist*)))
→ (¬ eval (make-disjunct (append (*flist*, *auxlist*)),
 falsify-taut (*flist*, *auxlist*)))

Definition: tautologyp (*flist*) = tautologyp1 (*flist*, **nil**)

Definition: taut-proof (*flist*) = taut-proof1 (*flist*, **nil**)

Event: Disable append; name this event 'g0300'.

Theorem: form-list-append-nil
make-disjunct (append (*flist*, **nil**)) = make-disjunct (*flist*)

Theorem: tautology-theorem
(form-list (*flist*) ∧ tautologyp (*flist*) ∧ (*concl* = make-disjunct (*flist*)))
→ proves (taut-proof (*flist*), *concl*)

Theorem: taut-eval2
(tautologyp1 (*flist*, *auxlist*)
 ∧ (*concl* = make-disjunct (append (*flist*, *auxlist*))))
→ eval (*concl*, *alist*)

Theorem: tautologies-are-true
(form-list (*flist*) ∧ tautologyp (*flist*))
→ eval (make-disjunct (*flist*), *alist*)

Theorem: not-taut-falsify2
(form-list (*flist*)
 ∧ prop-atomp-list (*auxlist*)
 ∧ (¬ eval (make-disjunct (*auxlist*), falsify (*auxlist*)))
 ∧ (¬ tautologyp1 (*flist*, *auxlist*))
 ∧ (*concl* = make-disjunct (append (*flist*, *auxlist*))))
→ (¬ eval (*concl*, falsify-taut (*flist*, *auxlist*)))

Theorem: truths-are-tautologies
(form-list (*flist*) ∧ (¬ tautologyp (*flist*)))
→ (¬ eval (make-disjunct (*flist*), falsify-taut (*flist*, **nil**)))

Event: Enable truths-are-tautologies; name this event 'g2439'.

Event: Enable not-taut-falsify2; name this event 'g2440'.

EVENT: Enable tautologies-are-true; name this event 'g2441'.

EVENT: Enable taut-eval2; name this event 'g2442'.

EVENT: Enable form-list-append-nil; name this event 'g2443'.

EVENT: Enable taut-proof; name this event 'g2444'.

EVENT: Enable not-taut-false; name this event 'g2445'.

EVENT: Enable not-falsify-taut; name this event 'g2446'.

EVENT: Enable append-nlistp; name this event 'g2447'.

EVENT: Enable falsify-taut; name this event 'g2448'.

EVENT: Enable formula-cases; name this event 'g2449'.

EVENT: Enable formula-cases1; name this event 'g2450'.

EVENT: Enable prop-atomp-falsify; name this event 'g2451'.

EVENT: Enable prop-atomp-auxlist; name this event 'g2453'.

EVENT: Enable prop-atomp-list; name this event 'g2454'.

EVENT: Enable falsify-step; name this event 'g2455'.

EVENT: Enable falsify; name this event 'g2456'.

EVENT: Enable not-eval-prop-atomp; name this event 'g2457'.

EVENT: Enable taut-eval; name this event 'g2458'.

EVENT: Enable eval-dble-neg-type; name this event 'g2459'.

EVENT: Enable eval-nor-type; name this event ‘g2460’.

EVENT: Enable eval-or-type; name this event ‘g2461’.

EVENT: Enable eval-prop-atomp; name this event ‘g2463’.

EVENT: Enable neg-list-eval; name this event ‘g2464’.

EVENT: Enable eval-neg-elem-form; name this event ‘g2465’.

EVENT: Enable elem-form-eval; name this event ‘g2471’.

EVENT: Enable eval; name this event ‘g2472’.

THEOREM: eval-tautologyp
(form-list (*flist*) $\wedge$ eval (make-disjunct (*flist*), falsify-taut (*flist*, **nil**)))
$\rightarrow$ tautologyp (*flist*)

DEFINITION:
lis-not (*flist*)
= **if** *flist* $\simeq$ **nil then nil**
else cons (f-not (car (*flist*)), lis-not (cdr (*flist*))) **endif**

DEFINITION:
taut-conseq (*flist*, *exp*) = tautologyp (append (lis-not (*flist*), cons (*exp*, **nil**)))

DEFINITION:
tautconseq-proof (*flist*, *exp*, *pflist*)
= list-detach-proof (*flist*,
exp,
pflist,
taut-proof (append (lis-not (*flist*), cons (*exp*, **nil**))))

THEOREM: list-implies-reduc
list-implies (*flist*, *exp*)
= make-disjunct (append (lis-not (*flist*), cons (*exp*, **nil**)))

THEOREM: append-exp-form-list
(form-list (*flist*) $\wedge$ formula (*exp*, **t**, 0))
$\rightarrow$ form-list (append (lis-not (*flist*), cons (*exp*, **nil**)))

THEOREM: taut-conseq-proves
(form-list (*flist*)
 ∧ formula (*exp*, **t**, 0)
 ∧ taut-conseq (*flist*, *exp*)
 ∧ proves-list (*pflist*, *flist*))
→ proves (tautconseq-proof (*flist*, *exp*, *pflist*), *exp*)

EVENT: Enable tautconseq-proof; name this event 'g0276'.

THEOREM: eval-tautconseq
(form-list (*flist*)
 ∧ formula (*exp*, **t**, 0)
 ∧ eval (make-disjunct (append (lis-not (*flist*), cons (*exp*, **nil**))),
 falsify-taut (append (lis-not (*flist*), cons (*exp*, **nil**)), **nil**)))
→ taut-conseq (*flist*, *exp*)

EVENT: Enable taut-conseq; name this event 'g0277'.

EVENT: Enable falsify-taut; name this event 'g0282'.

EVENT: Enable formula; name this event 'g0295'.

THEOREM: eval-tautconseq-proof-proves
(eval (make-disjunct (append (lis-not (*flist*), cons (*exp*, **nil**))),
 falsify-taut (append (lis-not (*flist*), cons (*exp*, **nil**)), **nil**))
 ∧ proves-list (*pflist*, *flist*)
 ∧ form-list (*flist*)
 ∧ formula (*exp*, **t**, 0))
→ proves (tautconseq-proof (*flist*, *exp*, *pflist*), *exp*)

EVENT: Enable taut-conseq-proves; name this event 'g0283'.

DEFINITION:
f-iff-reduc-proof (*a*, *b*, *pf1*, *pf2*)
= tautconseq-proof (list (f-iff (*a*, *b*), *a*), *b*, list (*pf1*, *pf2*))

Index