lib/basic/sets.ath

```
load "nat-minus"

module Set {

structure (Set S) :=  null | (insert S (Set S))

define (lst->set L) :=
   (let ((f ((from-list "(Set.Set 'S)" id) lst->set)))
     (f L))

define (lst->set L) :=
  match L {
    [] => null
  | (list-of x rest) => (insert (lst->set x) (lst->set rest))
  | _ => L
  }

(lst->set [1 2 3])

define (set->lst S) :=
 (let ((f ((to-list "(Set.Set 'T)" dedup) set->lst)))
    (f S))

define (set->lst-aux s) :=
  match s {
    null => []
  | (insert x rest) => (add (set->lst-aux x) (set->lst-aux rest))
  | _ => s
  }

define (set->lst s) :=
  match (set->lst-aux s) {
    (some-list L) => (dedup L)
  | _ => s
  }

#define (set->lst S) :=
# (let ((f ((to-list "(Set 'T)" dedup) set->lst)))
#     (f S))

#define set->lst := ((to-list "(Set.Set 'T)" dedup) id)

(set->lst (1 insert 2 insert 1 insert 3 insert null))

expand-input insert [id lst->set]

define ++ := insert

 (1 ++ [2 3])

set-precedence ++ 210

define [x y z h h' a b s s' t t' s1 s2 s3 A B C D E U] :=
       [?x ?y ?z ?h ?h' ?a ?b ?s:(Set 'T1) ?s':(Set 'T2)
        ?t:(Set 'T3) ?t':(Set 'T4) ?s1:(Set 'T5)
        ?s2:(Set 'T6) ?s3:(Set 'T7) ?A:(Set 'T8)
        ?B:(Set 'T9) ?C:(Set 'T10)
        ?D:(Set 'T10) ?E:(Set 'T11) ?U]

declare in: (T) [T (Set T)] -> Boolean [[id lst->set]]

assert* in-def :=
  [(~ _ in null)
   (x in h ++ t <==> x = h | x in t)]

(eval 23 in [1 5 23 98])

(eval 23 in [1 5 98])
```

```

(eval 5 in [])

(eval 5 in [5])

conclude null-characterization := (forall x . x in [] <==> false)
  pick-any x
    (!equiv
      assume hyp := (x in [])
        (!absurd hyp
                (!chain-> [true ==> (~ x in []) [in-def]]))
      assume false
        (!from-false (x in [])))

conclude in-lemma-1 := (forall x A . x in x ++ A)
  pick-any x A
    (!chain-> [(x = x) ==> (x in x ++ A) [in-def]])


define NC := null-characterization

declare singleton: (T) [T] -> (Set T)

assert* singleton-axiom := (singleton x = x ++ null)

conclude singleton-characterization :=
  (forall x y . x in singleton y <==> x = y)
 pick-any x y
  (!chain [(x in singleton y)
      <==> (x in y ++ null) [singleton-axiom]
      <==> (x = y | x in null)    [in-def]
      <==> (x = y | false)         [null-characterization]
      <==> (x = y)                 [prop-taut]])

define singleton-lemma := (forall x . x in singleton x)
 pick-any x
    (!chain-> [(x = x)
           ==> (x in singleton x)  [singleton-characterization]])

declare subset, proper-subset: (S) [(Set S) (Set S)] -> Boolean [[lst->set lst->set]]

assert* subset-def :=
  [([] subset _)
   (h ++ t subset A <==> h in A & t subset A)]

(eval [1 2] subset [3 2 4 1 5])

(eval [1 2] subset [3 2])

(eval [] subset [])

define subset-characterization-1 :=
  by-induction (forall A B . A subset B ==> forall x . x in A ==> x in B) {
    null => pick-any B
             assume (null subset B)
               pick-any x
                 (!chain [(x in null) ==> false    [NC]
                                      ==> (x in B) [prop-taut]])
  | (A as (insert h t)) =>
      pick-any B
        assume hyp := (A subset B)
          pick-any x
            let {ih := (forall B . t subset B ==>
                                    forall x . x in t ==> x in B);
                 _ := (!chain-> [hyp ==> (t subset B) [subset-def]])}
             assume hyp' := (x in A)
               (!cases (!chain<- [(x = h | x in t) <== hyp' [in-def]])
                 assume (x = h)
                   (!chain-> [hyp ==>  (h in B)   [subset-def]
                                  ==>  (x in B)   [(x = h)]])
```

```
                  (!chain [(x in t) ==> (x in B)   [ih]]))
  }


define subset-characterization-2 :=
  by-induction (forall A B . (forall x . x in A ==> x in B) ==> A subset B) {
    null => pick-any B
              assume (forall x . x in null ==> x in B)
                (!chain-> [true ==> (null subset B) [subset-def]])
  | (A as (insert h t)) =>
      pick-any B
        assume hyp :=  (forall x . x in A ==> x in B)
          let  {ih := (forall B . (forall x . x in t ==> x in B)
                                   ==> t subset B);
                goal := (A subset B);
                ih-cond := pick-any x
                             (!chain [(x in t) ==> (x in A)  [in-def]
                                               ==> (x in B)  [hyp]]);
                _ := (!chain-> [ih-cond ==> (t subset B)    [ih]])}
            (!chain-> [(h = h)
                   ==> (h in A)              [in-def]
                   ==> (h in B)              [hyp]
                   ==> (h in B & t subset B) [augment]
                   ==> goal                  [subset-def]])
  }

conclude subset-characterization :=
  (forall s1 s2 . s1 subset s2 <==> forall x . x in s1 ==> x in s2)
     pick-any s1 s2
       (!equiv (!chain [(s1 subset s2)
                    ==> (forall x . x in s1 ==> x in s2) [subset-characterization-1]])
               (!chain [(forall x . x in s1 ==> x in s2)
                    ==> (s1 subset s2)                   [subset-characterization-2]]))

define SC := subset-characterization

define subset-intro :=
  method (p)
    match p {
      (forall (some-var x) ((x in (some-term A)) ==> (x in (some-term B)))) =>
        (!chain-> [p ==> (A subset B) [subset-characterization]])
    }

assert* set-identity :=
  (A = B <==> A subset B & B subset A)

(eval 1 ++ 2 ++ [] = 2 ++ 1 ++ [])

(eval 1 ++ 2 ++ 3 ++ 4 ++ [] = 3 ++ 2 ++ 1 ++ [])


conclude set-identity-characterization :=
  (forall A B . A = B <==> forall x . x in A <==> x in B)
 pick-any A:(Set 'S) B
   (!equiv
     assume hyp := (A = B)
       pick-any x
         let {_ := (!chain-> [hyp ==> (A subset B) [set-identity]]);
              _ := (!chain-> [hyp ==> (B subset A) [set-identity]])}
           (!chain [(x in A) <==> (x in B) [subset-characterization]])
     assume hyp := (forall x . x in A <==> x in B)
       let {A-subset-B := (!subset-intro
                            pick-any x
                              (!chain [(x in A) ==> (x in B) [hyp]]));
            B-subset-A := (!subset-intro
                            pick-any x
                             (!chain [(x in B) ==> (x in A) [hyp]]));
            p := (!both A-subset-B B-subset-A)}
          (!chain-> [p ==> (A = B) [set-identity]]))
```

```
define SIC := set-identity-characterization

define set-identity-intro :=
  method (p1 p2)
    match [p1 p2] {
      [(A subset B) (B subset A)] =>
        (!chain-> [p1 ==> (p1 & p2) [augment]
                      ==> (A = B)   [set-identity]])
    }

define set-identity-intro-direct :=
  method (premise)
    match premise {
      (forall x ((x in A) <==> (x in B))) =>
        (!chain-> [premise ==> (A = B) [set-identity-characterization]])
    }


assert* proper-subset-def :=
  [(s1 proper-subset s2 <==> s1 subset s2 & s1 =/= s2)]

(eval [1 2] proper-subset [2 3 1])

(eval [1 2] proper-subset [2 1])

conclude neg-set-identity-characterization-1 :=
  (forall s1 s2 . s1 =/= s2 <==> ~ s1 subset s2 | ~ s2 subset s1)
pick-any s1 s2
  (!chain [(s1 =/= s2)
      <==> (~ (s1 subset s2 & s2 subset s1)) [set-identity]
      <==> (~ s1 subset s2 | ~ s2 subset s1) [prop-taut]])

conclude neg-set-identity-characterization-2 :=
  (forall s1 s2 . s1 =/= s2 <==>
     (exists x . x in s1 & ~ x in s2) |
     (exists x . x in s2 & ~ x in s1))
pick-any s1 s2
  (!chain [(s1 =/= s2)
      <==> (~ s1 subset s2 | ~ s2 subset s1)  [neg-set-identity-characterization-1]
      <==> (~ (forall x . x in s1 ==> x in s2) | ~ (forall x . x in s2 ==> x in s1))       [SC]
      <==> ((exists x . ~ (x in s1 ==> x in s2)) | (exists x . ~ (x in s2 ==> x in s1)))   [qn]
      <==> ((exists x . x in s1 & ~ x in s2) | (exists x . x in s2 & ~ x in s1))           [prop-taut]])


define proper-subset-characterization :=
  (forall s1 s2 . s1 proper-subset s2 <==> s1 subset s2 & exists x . x in s2 & ~ x in s1)

conclude PSC := proper-subset-characterization
  pick-any s1 s2
     (!chain [(s1 proper-subset s2)
         <==> (s1 subset s2 & s1 =/= s2)   [proper-subset-def]
         <==> (s1 subset s2 & ((exists x . x in s1 & ~ x in s2) |
                               (exists x . x in s2 & ~ x in s1)))  [neg-set-identity-characterization-2]
         <==> (s1 subset s2 & (((s1 subset s2) & (exists x . x in s1 & ~ x in s2)) |
                                (exists x . x in s2 & ~ x in s1)))  [prop-taut]
         <==> (s1 subset s2 & (((forall x . x in s1 ==> x in s2) & (exists x . x in s1 & ~ x in s2)) |
                                 (exists x . x in s2 & ~ x in s1))) [SC]
         <==> (s1 subset s2 & ((~~ (forall x . x in s1 ==> x in s2) & (exists x . x in s1 & ~ x in s2)) |
                                 (exists x . x in s2 & ~ x in s1))) [bdn]
         <==> (s1 subset s2 & ((~ (exists x . ~ (x in s1 ==> x in s2)) & (exists x . x in s1 & ~ x in s2)) |
                                 (exists x . x in s2 & ~ x in s1))) [qn]
         <==> (s1 subset s2 & ((~ (exists x . x in s1 & ~ x in s2) & (exists x . x in s1 & ~ x in s2)) |
                                 (exists x . x in s2 & ~ x in s1))) [prop-taut]
         <==> (s1 subset s2 & (false |  (exists x . x in s2 & ~ x in s1)))  [prop-taut]
         <==> (s1 subset s2 & (exists x . x in s2 & ~ x in s1))  [prop-taut]])


conclude proper-subset-lemma :=
  (forall A B x . A subset B & x in B & ~ x in A ==> A proper-subset B)
    pick-any A B x
```

```
        assume h := (A subset B & x in B & ~ x in A)
          (!chain-> [(x in B)
                ==> (x in B & ~ x in A) [augment]
                ==> (exists x . x in B & ~ x in A) [existence]
                ==> (A subset B & exists x . x in B & ~ x in A) [augment]
                ==> (A proper-subset B) [PSC]])

conclude in-lemma-2 := (forall h t . h in t ==> h ++ t = t)
  pick-any h t
    assume hyp := (h in t)
      (!set-identity-intro-direct
         pick-any x
           (!chain [(x in h ++ t)
               <==> (x = h | x in t)    [in-def]
               <==> (x in t | x in t)   [hyp prop-taut]
               <==> (x in t)            [prop-taut]])))

conclude in-lemma-3 := (forall x h t . x in t ==> x in h ++ t)
  pick-any x h t
    (!chain [(x in t)
        ==> (x = h | x in t)  [alternate]
        ==> (x in h ++ t)     [in-def]])


conclude in-lemma-4 :=
  (forall A x y . x in A ==> y in A <==> y = x | y in A)
pick-any A x y
  assume (x in A)
    (!equiv assume h := (y in A)
              (!chain-> [h ==> (y = x | y in A) [alternate]])
            assume h := (y = x | y in A)
              (!cases h
                 (!chain [(y = x) ==> (y in A)  [(x in A)]])
                 (!chain [(y in A) ==> (y in A) [claim]])))

conclude null-characterization-2 :=
  (forall A . A = null <==> forall x . ~ x in A)
pick-any A
  (!chain [(A = null)
      <==> (forall x . x in A <==> x in null)   [SIC]
      <==> (forall x . x in A <==> false)       [NC]
      <==> (forall x . ~ x in A)                [prop-taut]])

define NC-2 := null-characterization-2

conclude NC-3 :=
  (forall A . A =/= null <==> exists x . x in A)
pick-any A
  (!chain [(A =/= null)
      <==> (~ forall x . ~ x in A) [NC-2]
      <==> (exists x . ~ ~ x in A) [qn-strict]
      <==> (exists x . x in A)     [bdn]])

define (non-empty S) := (S =/= null)

conclude subset-reflexivity := (forall A . A subset A)
  pick-any A
    (!subset-intro
       pick-any x
         (!chain [(x in A) ==> (x in A) [claim]]))

conclude subset-antisymmetry :=
  (forall A B . A subset B & B subset A ==> A = B)
 pick-any A B
   assume hyp := (A subset B & B subset A)
     (!set-identity-intro (A subset B) (B subset A))

conclude subset-transitivity :=
 (forall A B C . A subset B & B subset C ==> A subset C)
   pick-any A B C
```

```
     assume (A subset B & B subset C)
       (!subset-intro
          pick-any x
            (!chain [(x in A)
                 ==> (x in B) [subset-characterization]
                 ==> (x in C) [subset-characterization]]))


conclude subset-lemma-1 :=
  (forall A B x . A subset B & x in B ==> x ++ A subset B)
pick-any A B x
  assume hyp := (A subset B & x in B)
    (!subset-intro
      pick-any y
        (!chain [(y in x ++ A)
             ==> (y = x | y in A)   [in-def]
             ==> (y in B | y in A)  [(x in B)]
             ==> (y in B | y in B)  [SC]
             ==> (y in B)           [prop-taut]]))

conclude subset-lemma-2 :=
  (forall h t A . h ++ t subset A ==> t subset A)
pick-any h t A
  assume (h ++ t subset A)
    (!subset-intro
       pick-any x
         (!chain [(x in t)
              ==> (x = h | x in t)  [alternate]
              ==> (x in h ++ t)     [in-def]
              ==> (x in A)          [SC]]))



declare remove: (S) [(Set S) S] -> (Set S) [- [lst->set id]]

assert* remove-def :=
  [(null - _ = null)
   (h ++ t - x = t - x <== x = h)
   (h ++ t - x = h ++ (t - x) <== x =/= h)]

(eval [1 2 3 2 5] - 2)

conclude remove-characterization :=
  (forall A x y . y in A - x <==> y in A & y =/= x)
by-induction remove-characterization {
  null => pick-any x y
            (!chain [(y in null - x)
                 <==> (y in null)
                 <==> false
                 <==> (y in null & y =/= x)])
| (A as (insert h t)) =>
   let {IH := (forall x y . y in t - x <==> y in t & y =/= x)}
    pick-any x y
      (!two-cases
         assume case-1 := (x = h)
           (!chain [(y in A - x)
               <==> (y in t - x)                 [remove-def]
               <==> (y in t & y =/= x)           [IH]
               <==> ((y = x | y in t) & y =/= x) [prop-taut]
               <==> ((y = h | y in t) & y =/= x) [case-1]
               <==> (y in A & y =/= x)           [in-def]])
         assume case-2 := (x =/= h)
          let {lemma := assume (y = h)
                          (!chain-> [case-2 ==> (y =/= x) [(y = h)]])}
           (!chain [(y in A - x)
               <==> (y in h ++ (t - x))                    [remove-def]
               <==> (y = h | y in t - x)                   [in-def]
               <==> (y = h | (y in t & y =/= x))           [IH]
               <==> ((y = h | y in t) & (y = h | y =/= x)) [prop-taut]
               <==> (y in A & (y = h | y =/= x))           [in-def]
```

```
               <==> (y in A & (y =/= x | y =/= x))          [prop-taut lemma]
               <==> (y in A & y =/= x)                      [prop-taut]]))
}

conclude remove-corollary := (forall A x  . ~ x in A - x)
  pick-any A x
   (!by-contradiction (~ x in A - x)
      (!chain [(x in A - x)
           ==> (x in A & x =/= x)  [remove-characterization]
           ==> (x =/= x)           [right-and]
           ==> (x = x & x =/= x)   [augment]
           ==> false               [prop-taut]]))

conclude remove-corollary-2 :=
  (forall A x  . ~ x in A ==> A - x = A)
pick-any A x
  assume hyp := (~ x in A)
    (!set-identity-intro-direct
       pick-any y
         (!equiv
           (!chain [(y in A - x)
                ==> (y in A & y =/= x) [remove-characterization]
                ==> (y in A)           [left-and]])
           assume (y in A)
             let {_ := (!by-contradiction (y =/= x)
                          assume (y = x)
                            (!absurd (y in A)
                                       (!chain-> [hyp ==> (~ y in A) [(y = x)]])));
                  lemma := (!both (y in A) (y =/= x))}
               (!chain-> [lemma ==> (y in A - x) ])))

conclude remove-corollary-3 :=
  (forall A x y . x in A & y =/= x ==> x in A - y)
pick-any A x y
  assume hyp := (x in A & y =/= x)
    let {_ := (!ineq-sym (y =/= x))}
      (!chain-> [hyp ==> (x in A - y) [remove-characterization]])

conclude remove-corollary-4 :=
  (forall A x y  . ~ x in A ==> ~ x in A - y)
  pick-any A x y
    (!chain [( ~ x in A) ==> (~ x in A - y) [remove-characterization]])

conclude remove-corollary-5 :=
  (forall A B x . A subset B & ~ x in A ==> A subset B - x)
pick-any A B x
  assume h := (A subset B & ~ x in A)
   (!subset-intro
     pick-any y
       assume h2 := (in y A)
         let {_ := (!chain-> [h2 ==> (in y B) [SC]]);
              _ := (!by-contradiction (y =/= x)
                      assume (y = x)
                        (!absurd (in y A)
                                  (!chain-> [(~ x in A) ==> (~ y in A) [(y = x)]])));
              S := (!both (in y B) (y =/= x))}
          (!chain-> [S ==> (y in B - x) [remove-characterization]]))


conclude remove-corollary-6 := (forall A h t . A subset h ++ t ==> A - h subset t)
pick-any A:(Set.Set 'S) h:'S t:(Set.Set 'S)
 assume hyp := (A subset h ++ t)
   (!subset-intro
     pick-any x
       assume hyp' := (x in A - h)
         let {_ := (!chain-> [hyp' ==> (x in A & x =/= h) [remove-characterization]]);
             disj := (!chain-> [(x in A) ==> (x in h ++ t)     [SC]
                                         ==> (x = h | x in t) [in-def]])}
           (!cases disj
              (!chain [(x = h)
```

```
                      ==> (x = h & x =/= h) [augment]
                      ==> false             [prop-taut]
                      ==> (x in t)          [prop-taut]])
               (!chain [(x in t) ==> (x in t) [claim]]))))


conclude remove-corollary-7 := (forall A x . A - x subset A)
pick-any A:(Set.Set 'S) x:'S
  (!subset-intro
     pick-any y
       (!chain [(y in A - x)
            ==> (y in A)      [remove-characterization]]))


conclude remove-corollary-8 :=
  (forall A x . x in A ==> A = x ++ (A - x))
pick-any A:(Set.Set 'S) x:'S
  assume (x in A)
    let {p1 := (!subset-intro
                 pick-any y:'S
                   assume (y in A)
                     (!two-cases
                       assume (x = y)
                         (!chain-> [true ==> (x in x ++ (A - x)) [in-lemma-1]
                                         ==> (y in x ++ (A - x)) [(x = y)]])
                       assume (x =/= y)
                         (!chain-> [(x =/= y)
                                ==> (y in A & x =/= y)  [augment]
                                ==> (y in A - x)        [remove-corollary-3]
                                ==> (y in x ++ (A - x)) [in-def]])));
         p2 := (!subset-intro
                 pick-any y:'S
                  assume hyp := (y in x ++ (A - x))
                     (!cases (!chain<- [(y = x | y in A - x) <== hyp [in-def]])
                        assume (y = x)
                          (!chain-> [(y = x) ==> (y in A) [(x in A)]])
                        assume (y in A - x)
                          (!chain-> [(y in A - x) ==> (y in A) [remove-characterization]])))}
      (!set-identity-intro p1 p2)

conclude subset-lemma-3 :=
  (forall A t h . A subset h ++ t & h in A ==> exists B . B subset t & A = h ++ B)
pick-any A:(Set.Set 'S) t h:'S
  assume hyp := (A subset h ++ t & h in A)
    let {p := (!chain-> [(A subset h ++ t) ==> (A - h subset t) [remove-corollary-6]])}
      (!chain-> [(h in A)
            ==> (A = h ++ (A - h))                    [remove-corollary-8]
            ==> (p & A = h ++ (A - h))                [augment]
            ==> (exists B . B subset t & A = h ++ B) [existence]])


conclude subset-lemma-4 :=
  (forall A h t . ~ h in A & A subset h ++ t ==> A subset t)
pick-any A h t
  assume hyp := (~ h in A & A subset h ++ t)
    (!subset-intro
      pick-any x
       assume (x in A)
         (!cases (!chain<- [(x = h | x in t) <== (x in h ++ t) [in-def]
                                             <== (x in A)      [SC]])
            (!chain [(x = h)
                 ==> (~ x in A) [(~ h in A)]
                 ==> (x in A & ~ x in A) [augment]
                 ==> (in x t)    [prop-taut]])
            (!chain [(x in t) ==> (x in t) [claim]])))


conclude subset-lemma-5 :=
 (forall A t h . A subset t ==> A subset h ++ t)
pick-any A t h
```

```
  assume hyp := (A subset t)
    (!subset-intro
       pick-any x
         (!chain [(x in A) ==> (x in t)      [SC]
                           ==> (x in h ++ t) [in-def]]))

conclude subset-lemma-6 :=
  (forall A . A subset null <==> A = null)
pick-any A
  (!equiv assume (A subset null)
            (!by-contradiction (A = null)
              assume (A =/= null)
                pick-witness x for (!chain<- [(exists x . x in A) <== (A =/= null) [NC-3]])
                  (!chain-> [(x in A) ==> (x in null) [SC]
                                      ==> false       [NC]]))
          assume (A = null)
            (!chain-> [true ==> (A subset A)    [subset-reflexivity]
                            ==> (A subset null) [(A = null)]])))


conclude subset-lemma-7 :=
  (forall A B x . ~ x in A & B subset A ==> ~ x in B)
pick-any A B x
  assume hyp := (~ x in A & B subset A)
    (!by-contradiction (~ x in B)
      (!chain [(x in B) ==> (x in A)            [SC]
                        ==> (x in A & ~ x in A) [augment]
                        ==> false               [prop-taut]]))



declare union, intersection, diff: (S) [(Set S) (Set S)] -> (Set S) [120 [lst->set lst->set]]

define [\/ /\ \] := [union intersection diff]

assert* union-def :=
 [([] \/ s = s)
  (h ++ t \/ s = h ++ (t \/ s))]

transform-output eval [set->lst]

(eval [1 2 3] \/ [4 5 6])

(eval [1 2] \/ [1 2])

conclude union-characterization-1 :=
  (forall A B x . x in A \/ B ==> x in A | x in B)
 by-induction union-characterization-1 {
   null => pick-any B x
            (!chain [(x in null \/ B)
                 ==> (x in B)             [union-def]
                 ==> (x in null | x in B) [alternate]])
 | (A as (h insert t)) =>
    let {IH := (forall B x . x in t \/ B ==> x in t | x in B)}
      pick-any B x
        (!chain [(x in A \/ B)
             ==> (x in h ++ (t \/ B))        [union-def]
             ==> (x = h | x in t \/ B)       [in-def]
             ==> (x = h | x in t | x in B)   [IH]
             ==> ((x = h | x in t) | x in B) [prop-taut]
             ==> (x in A | x in B)           [in-def]])
  }

conclude union-characterization-2 :=
 (forall A B x . x in A | x in B ==> x in A \/ B)
   by-induction union-characterization-2 {
    (A as null) =>
       pick-any B x
        (!chain [(x in null | x in B)
             ==> (false | x in B)     [NC]
```

```
              ==> (x in B)             [prop-taut]
              ==> (x in null \/ B)     [union-def]])

 | (A as (insert h t)) =>
     pick-any B x
       let {IH := (forall B x . x in t | x in B ==> x in t \/ B)}
          (!chain [(x in A | x in B)
               ==> ((x = h | x in t) | x in B)   [in-def]
               ==> (x = h | (x in t | x in B))   [prop-taut]
               ==> (x = h | x in t \/ B)         [IH]
               ==> (x in h ++ (t \/ B))          [in-def]
               ==> (x in A \/ B)                 [union-def]])
  }


conclude union-characterization :=
 (forall A B x . x in A \/ B <==> x in A | x in B)
  pick-any A B x
    (!chain [(x in A \/ B)
        <==> (x in A | x in B) [union-characterization-1
                                union-characterization-2]])


define UC := union-characterization

assert* intersection-def :=
 [(null /\ s = null)
  (h ++ t /\ A = h ++ (t /\ A) <== h in A)
  (h ++ t /\ A = t /\ A <== ~ h in A)]

(eval [1 2 1] /\ [5 1 3])

(eval [1 2 1] /\ [5])

conclude intersection-characterization-1 :=
 (forall A B x . x in A /\ B ==> x in A & x in B)
by-induction intersection-characterization-1 {
  null => pick-any B x
            (!chain [(x in null /\ B)
                 ==> (x in null)          [intersection-def]
                 ==> false                [NC]
                 ==> (x in null & x in B) [prop-taut]])
| (A as (insert h t)) =>
  let {IH := (forall B x . x in t /\ B ==> x in t & x in B)}
    pick-any B x
      (!two-cases
        assume (h in B)
          (!chain [(x in (h ++ t) /\ B)
                ==> (x in h ++ (t /\ B))                  [intersection-def]
                ==> (x = h | x in t /\ B)                 [in-def]
                ==> (x = h | x in t & x in B)             [IH]
                ==> ((x = h | x in t) & (x = h | x in B)) [prop-taut]
                ==> (x in A & (x in B | x in B))          [in-def (h in B)]
                ==> (x in A & x in B)                     [prop-taut]
                  ])
        assume (~ h in B)
          (!chain [(x in A /\ B)
                ==> (x in t /\ B)      [intersection-def]
                ==> (x in t & x in B) [IH]
                ==> (x in A & x in B) [in-def]]))
    }

conclude intersection-characterization-2 :=
 (forall A B x . x in A & x in B ==> x in A /\ B)
by-induction intersection-characterization-2 {
  (A as null) =>
    pick-any B x
      (!chain [(x in null & x in B)
           ==> (x in null)          [left-and]
           ==> false                [NC]
```

```
           ==> (x in null /\ B)      [prop-taut]])
| (A as (insert h t)) =>
  let {IH := (forall B x . x in t & x in B ==> x in t /\ B)}
    pick-any B x
      (!two-cases
        assume (h in B)
          (!chain [(x in A & x in B)
               ==> ((x = h | x in t) & x in B)             [in-def]
               ==> ((x = h & x in B) | (x in t & x in B)) [prop-taut]
               ==> (x = h | x in t & x in B)              [prop-taut]
               ==> (x = h | x in t /\ B)                  [IH]
               ==> (x in h ++ (t /\ B))                   [in-def]
               ==> (x in A /\ B)                      [intersection-def]])
        assume case2 := (~ h in B)
          (!chain [(x in A & x in B)
               ==> ((x = h | x in t) & x in B)                [in-def]
               ==> ((~ x in B | x in t) & x in B)             [case2]
               ==> ((~ x in B & x in B) | (x in t & x in B)) [prop-taut]
               ==> (false | x in t & x in B)                 [prop-taut]
               ==> (x in t & x in B)                         [prop-taut]
               ==> (x in t /\ B)                             [IH]
               ==> (x in A /\ B)                      [intersection-def]]))
}


conclude intersection-characterization :=
 (forall A B x . x in A /\ B <==> x in A & x in B)
  pick-any A B x
    (!equiv
       (!chain [(x in A /\ B)
            ==> (x in A & x in B) [intersection-characterization-1]])
       (!chain [(x in A & x in B)
            ==> (x in A /\ B)     [intersection-characterization-2]]))

define IC := intersection-characterization

conclude intersection-subset-theorem :=
  (forall A B . A /\ B subset A)
pick-any A B
  (!subset-intro
     pick-any x
       (!chain [(x in A /\ B)
            ==> (x in A)       [IC]]))

assert* diff-def :=
 [(null \ _ = null)
  (h ++ t \ A = t \ A <== h in A)
  (h ++ t \ A = h ++ (t \ A) <== ~ h in A)]

(eval [1 2 3] \ [3 1])

conclude diff-characterization-1 :=
  (forall A B x . x in A \ B ==> x in A & ~ x in B)
 by-induction diff-characterization-1 {
      (A as null) =>
         pick-any B x
           (!chain [(x in A \ B)
                ==> (x in null)            [diff-def]
                ==> false                  [null-characterization]
                ==> (x in null & ~ x in B) [prop-taut]])
 | (A as (insert h t)) =>
     pick-any B x
       let {ih := (forall B x . x in t \ B ==> x in t & ~ x in B)}
        assume hyp := (x in A \ B)
          (!two-cases
            assume case1 := (h in B)
              (!chain-> [hyp
                    ==> (x in t \ B)          [diff-def]
                    ==> (x in t & ~ x in B)   [ih]
                    ==> (x in A & ~ x in B)   [in-def]])
```

```
            assume case2 := (~ h in B)
              (!cases (!chain<- [(x = h | x in t \ B)
                             <== (x in h ++ (t \ B)) [in-def]
                             <== hyp                 [diff-def]])
                assume case2-1 := (x = h)
                 (!chain-> [(h = h)
                        ==> (h in h ++ t)            [in-def]
                        ==> (h in h ++ t & ~ h in B) [augment]
                        ==> (x in A & ~ x in B)      [case2-1]])
                assume case2-2 := (x in t \ B)
                 (!chain-> [case2-2
                        ==> (x in t & ~ x in B)      [ih]
                        ==> (x in h ++ t & ~ x in B) [in-def]])))
     }

conclude diff-characterization-2 :=
  (forall A B x . x in A & ~ x in B ==> x in A \ B)
 by-induction diff-characterization-2 {
   (A as null) =>
      pick-any B x
        (!chain [(x in A & ~ x in B)
             ==> (x in A)             [left-and]
             ==> false                [null-characterization]
             ==> (x in A \ B)         [prop-taut]])
 | (A as (h insert t)) =>
     pick-any B x
       assume hyp := (x in A & ~ x in B)
         let {ih := (forall B x . x in t & ~ x in B ==> x in t \ B)}
          (!cases (!chain-> [(x in A) ==> (x = h | x in t) [in-def]])
             assume case-1 := (x = h)
               (!chain<- [(x in A \ B)
                      <== (x in h ++ (t \ B))  [diff-def case-1]
                      <== (h in h ++ (t \ B))  [case-1]
                      <== true                 [in-lemma-1]])
             assume case-2 := (x in t)
               (!two-cases
                  assume (h in B)
                    (!chain<- [(x in A \ B)
                           <== (x in t \ B)        [diff-def]
                           <== case-2              [ih]])
                  assume (~ h in B)
                    (!chain<- [(x in A \ B)
                           <== (x in h ++ (t \ B)) [diff-def]
                           <== (x in t \ B)        [in-def]
                           <== case-2              [ih]])))
  }

conclude diff-characterization :=
  (forall A B x . x in A \ B <==> x in A & ~ x in B)
  pick-any A B x
    (!equiv
       (!chain [(x in A \ B)
            ==> (x in A & ~ x in B) [diff-characterization-1]])
       (!chain [(x in A & ~ x in B)
            ==> (x in A \ B)        [diff-characterization-2]]))

define DC := diff-characterization


conclude intersection-commutes := (forall A B . A /\ B = B /\ A)
 pick-any A B
   (!set-identity-intro-direct
     pick-any x
       (!chain [(x in A /\ B) <==> (x in A & x in B)  [IC]
                              <==> (x in B & x in A)  [prop-taut]
                              <==> (x in B /\ A)      [IC]]))



conclude intersection-commutes := (forall A B . A /\ B = B /\ A)
```

```
 pick-any A B
   let {A-subset-of-B :=
         (!subset-intro
           pick-any x
            (!chain [(x in A /\ B)
                 ==> (x in A & x in B) [IC]
                 ==> (x in B & x in A) [prop-taut]
                 ==> (x in B /\ A)     [IC]]));
        B-subset-of-A :=
         (!subset-intro
            pick-any x
              (!chain [(x in B /\ A)
                   ==> (x in B & x in A) [IC]
                   ==> (x in A & x in B) [prop-taut]
                   ==> (x in A /\ B)     [IC]]))
        }
      (!set-identity-intro A-subset-of-B B-subset-of-A)

conclude intersection-commutes := (forall A B . A /\ B = B /\ A)
  let {M := method (A B) # derive (A /\ B subset B /\ A)
              (!subset-intro
                 pick-any x
                   (!chain [(x in A /\ B)
                        ==> (x in A & x in B) [IC]
                        ==> (x in B & x in A) [prop-taut]
                        ==> (x in B /\ A)     [IC]]))}
    pick-any A B
      (!set-identity-intro (!M A B) (!M B A))

conclude intersection-subset-theorem-2 :=
  (forall A B . A /\ B subset B)
pick-any A B
  (!chain-> [true ==> (B /\ A subset B)  [intersection-subset-theorem]
                  ==> (A /\ B subset B)  [intersection-commutes]])

conclude intersection-subset-theorem' :=
  (forall A B C . A subset B /\ C <==> A subset B & A subset C)
pick-any A B C
  (!equiv assume (A subset B /\ C)
            (!both (!subset-intro
                      pick-any x
                        (!chain [(x in A) ==> (x in B /\ C) [SC]
                                          ==> (x in B)      [IC]]))
                   (!subset-intro
                      pick-any x
                        (!chain [(x in A) ==> (x in B /\ C) [SC]
                                          ==> (x in C)      [IC]])))
          assume (A subset B & A subset C)
            (!subset-intro
              pick-any x
               assume (x in A)
                 let {_ := (!chain-> [(x in A) ==> (x in B) [SC]]);
                      _ := (!chain-> [(x in A) ==> (x in C) [SC]]);
                      p := (!both (x in B) (x in C))}
                   (!chain-> [p ==> (x in B /\ C) [IC]])))

conclude union-subset-theorem :=
  (forall A B C . A subset B | A subset C ==> A subset B \/ C)
pick-any A B C
  assume hyp := (A subset B | A subset C)
    (!cases hyp
      assume (A subset B)
        (!subset-intro
          pick-any x
            (!chain [(x in A) ==> (x in B)          [SC]
                              ==> (x in B | x in C) [alternate]
                              ==> (x in B \/ C)     [UC]]))
      assume (A subset C)
        (!subset-intro
          pick-any x
```

```
            (!chain [(x in A) ==> (x in C)          [SC]
                              ==> (x in B | x in C) [alternate]
                              ==> (x in B \/ C)     [UC]])))

conclude union-commutes := (forall A B . A \/ B = B \/ A)
 pick-any A B
   (!set-identity-intro-direct
      pick-any x
        (!chain [(x in A \/ B) <==> (x in A | x in B)  [UC]
                               <==> (x in B | x in A)  [prop-taut]
                               <==> (x in B \/ A)      [UC]]))

conclude intersection-associativity :=
  (forall A B C . A /\ (B /\ C) = (A /\ B) /\ C)
 pick-any A B C
   (!set-identity-intro-direct
     pick-any x
       (!chain [(x in A /\ B /\ C)
           <==> (x in A & x in B /\ C)       [IC]
           <==> (x in A & x in B & x in C)   [IC]
           <==> ((x in A & x in B) & x in C) [prop-taut]
           <==> ((x in A /\ B) & x in C)     [IC]
           <==> (x in (A /\ B) /\ C)         [IC]]))

conclude union-associativity :=
   (forall A B C . A \/ B \/ C = (A \/ B) \/ C)
 pick-any A B C
   (!set-identity-intro-direct
      pick-any x
        (!chain [(x in A \/ B \/ C)
            <==> (x in A | x in B \/ C)       [UC]
            <==> (x in A | x in B | x in C)   [UC]
            <==> ((x in A | x in B) | x in C) [prop-taut]
            <==> (x in A \/ B | x in C)       [UC]
            <==> (x in (A \/ B) \/ C)         [UC]]))

conclude /\-idempotence :=
  (forall A . A /\ A = A)
pick-any A
 (!set-identity-intro-direct
   pick-any x
     (!chain [(x in A /\ A)
         <==> (x in A & x in A)  [IC]
         <==> (x in A)           [prop-taut]]))

conclude \/-idempotence :=
  (forall A . A \/ A = A)
pick-any A
  (!set-identity-intro-direct
    pick-any x
     (!chain [(x in A \/ A)
         <==> (x in A | x in A)  [UC]
         <==> (x in A)           [prop-taut]]))

conclude union-null-theorem :=
  (forall A B . A \/ B = null <==> A = null & B = null)
pick-any A B
  (!chain [(A \/ B = null)
      <==> (forall x . x in A \/ B <==> x in null)       [SIC]
      <==> (forall x . x in A \/ B <==> false)           [NC]
      <==> (forall x . x in A | x in B <==> false)       [UC]
      <==> (forall x . ~ x in A & ~ x in B)              [prop-taut]
      <==> ((forall x . ~ x in A) & (forall x ~ x in B)) [taut]
      <==> (A = null & B = null)                         [NC-2]])


conclude distributivity-1 :=
  (forall A B C . A \/ (B /\ C) = (A \/ B) /\ (A \/ C))
  pick-any A B C
    (!set-identity-intro-direct
```

```
      pick-any x
        (!chain [(x in A \/ (B /\ C))
            <==> (x in A | x in B /\ C)                 [UC]
            <==> (x in A | x in B & x in C)             [IC]
            <==> ((x in A | x in B) & (x in A | x in C)) [prop-taut]
            <==> (x in A \/ B & x in A \/ C)            [UC]
            <==> (x in (A \/ B) /\ (A \/ C))            [IC]]))


conclude distributivity-2 :=
  (forall A B C . A /\ (B \/ C) = (A /\ B) \/ (A /\ C))
    pick-any A B C
      (!set-identity-intro-direct
       pick-any x
         (!chain [(x in A /\ (B \/ C))
             <==> (x in A & x in B \/ C)                 [IC]
             <==> (x in A & (x in B | x in C))           [UC]
             <==> ((x in A & x in B) | (x in A & x in C)) [prop-taut]
             <==> (x in A /\ B | x in A /\ C)            [IC]
             <==> (x in (A /\ B) \/ (A /\ C))            [UC]]))

conclude diff-theorem-1 := (forall A . A \ A = null)
  pick-any A
    (!set-identity-intro-direct
      pick-any x
       (!chain [(x in A \ A)
           <==> (x in A & ~ x in A) [DC]
           <==> false               [prop-taut]
           <==> (x in null)         [NC]]))

conclude diff-theorem-2 :=
  (forall A B C . B subset C ==> A \ C subset A \ B)
pick-any A B C
  assume (B subset C)
    (!subset-intro
      pick-any x
       (!chain [(x in A \ C)
            ==> (x in A & ~ x in C) [DC]
            ==> (x in A & ~ x in B) [SC]
            ==> (x in A \ B)        [DC]]))


define p := (forall A B C . B subset C ==> A \ B subset A \ C)

(falsify p 20)

conclude diff-theorem-3 :=
  (forall A B . A \ (A /\ B) = A \ B)
    pick-any A B
      (!set-identity-intro-direct
       pick-any x
         (!chain [(x in A \ (A /\ B))
             <==> (x in A & ~ x in A /\ B)                      [DC]
             <==> (x in A & ~ (x in A & x in B))                [IC]
             <==> (x in A & (~ x in A | ~ x in B))              [prop-taut]
             <==> ((x in A & ~ x in A) | (x in A & ~ x in B))  [prop-taut]
             <==> (false | x in A & ~ x in B)                   [prop-taut]
             <==> (x in A & ~ x in B)                           [prop-taut]
             <==> (x in A \ B)                                  [DC]]))

conclude diff-theorem-4 :=
  (forall A B . A /\ (A \ B) = A \ B)
    pick-any A B
      (!set-identity-intro-direct
       pick-any x
         (!chain [(x in A /\ (A \ B))
             <==> (x in A & x in A \ B)         [IC]
             <==> (x in A & x in A & ~ x in B) [DC]
             <==> (x in A & ~ x in B)           [prop-taut]
             <==> (x in A \ B)                  [DC]]))
```

```
conclude diff-theorem-5 :=
  (forall A B . (A \ B) \/ B = A \/ B)
    pick-any A B
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in (A \ B) \/ B)
              <==> (x in A \ B | x in B)                     [UC]
              <==> ((x in A & ~ x in B) | x in B)            [DC]
              <==> ((x in A | x in B) & (~ x in B | x in B)) [prop-taut]
              <==> ((x in A | x in B) & true)                [prop-taut]
              <==> (x in A | x in B)                         [prop-taut]
              <==> (x in A \/ B)                             [UC]]))

conclude diff-theorem-6 :=
  (forall A B . (A \/ B) \ B = A \ B)
    pick-any A B
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in (A \/ B) \ B)
              <==> (x in A \/ B & ~ x in B)                [DC]
              <==> ((x in A | x in B) & ~ x in B)          [UC]
              <==> (x in A & ~ x in B | x in B & ~ x in B) [prop-taut]
              <==> (x in A & ~ x in B | false)             [prop-taut]
              <==> (x in A \ B | false)                    [DC]
              <==> (x in A \ B)                            [prop-taut]]))

conclude diff-theorem-7 :=
  (forall A B . (A /\ B) \ B = null)
    pick-any A B
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in (A /\ B) \ B)
              <==> (x in A /\ B & ~ x in B)       [DC]
              <==> ((x in A & x in B) & ~ x in B) [IC]
              <==> false                          [prop-taut]
              <==> (x in null)                    [NC]]))

conclude diff-theorem-8 :=
  (forall A B . (A \ B) /\ B = null)
    pick-any A B
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in (A \ B) /\ B)
              <==> (x in A \ B & x in B)          [IC]
              <==> ((x in A & ~ x in B) & x in B) [DC]
              <==> false                          [prop-taut]
              <==> (x in null)                    [NC]]))

conclude diff-theorem-8 :=
  (forall A B C . A \ (B \/ C) = (A \ B) /\ (A \ C))
    pick-any A B C
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in A \ (B \/ C))
              <==> (x in A & ~ x in B \/ C)                      [DC]
              <==> (x in A & ~ (x in B | x in C))                [UC]
              <==> (x in A & ~ x in B & ~ x in C)                [prop-taut]
              <==> ((x in A & ~ x in B) & (x in A & ~ x in C))   [prop-taut]
              <==> (x in A \ B & x in A \ C)                     [DC]
              <==> (x in (A \ B) /\ (A \ C))                     [IC]]))

conclude diff-theorem-9 :=
  (forall A B C . A \ (B /\ C) = (A \ B) \/ (A \ C))
    pick-any A B C
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in A \ (B /\ C))
              <==> (x in A & ~ x in B /\ C)                      [DC]
              <==> (x in A & ~ (x in B & x in C))                [IC]
```

```
              <==> (x in A & (~ x in B | ~ x in C))            [prop-taut]
              <==> ((x in A & ~ x in B) | (x in A & ~ x in C)) [prop-taut]
              <==> (x in A \ B | x in A \ C)                   [DC]
              <==> (x in (A \ B) \/ (A \ C))                   [UC]]))

conclude diff-theorem-10 := (forall A B . A \ (A \ B) = A /\ B)
  pick-any A B
    (!set-identity-intro-direct
      pick-any x
        (!chain [(x in A \ (A \ B))
             <==> (x in A & ~ x in A \ B)                   [DC]
             <==> (x in A & ~ (x in A & ~ x in B))          [DC]
             <==> (x in A & (~ x in A | ~ ~ x in B))        [prop-taut]
             <==> ((x in A & ~ x in A) | (x in A & x in B)) [prop-taut]
             <==> (false | x in A & x in B)                 [prop-taut]
             <==> (x in A & x in B)                         [prop-taut]
             <==> (x in A /\ B)                             [IC]]))

conclude diff-theorem-11 := (forall A B . A subset B ==> A \/ (B \ A) = B)
  pick-any A B
    assume hyp := (A subset B)
      (!set-identity-intro-direct
        pick-any x
          (!chain
            [(x in A \/ (B \ A))
        <==> (x in A | x in B \ A)                       [UC]
        <==> (x in A | x in B & ~ x in A)                [DC]
        <==> ((x in A | x in B) & (x in A | ~ x in A))   [prop-taut]
        <==> (x in A | x in B)                           [prop-taut]
        <==> (x in B | x in B)                           [SC prop-taut]
        <==> (x in B)                                    [prop-taut]]))


conclude diff-theorem-12 :=
  (forall A B . A = (A \ B) \/ (A /\ B))
pick-any A B
  (!comm
    (!set-identity-intro-direct
       pick-any x
        (!chain [(x in (A \ B) \/ (A /\ B))
             <==> (x in A \ B | x in A /\ B)             [UC]
             <==> (x in A & ~ x in B | x in A & x in B)  [DC IC]
             <==> (x in A)                               [prop-taut]])))

conclude diff-theorem-13 :=
  (forall A B . (A \ B) /\ (A /\ B) = null)
pick-any A B
  (!set-identity-intro-direct
    pick-any x
      (!chain [(x in (A \ B) /\ (A /\ B))
          <==> (x in (A \ B) & x in A /\ B)              [IC]
          <==> ((x in A & ~ x in B) & (x in A & x in B)) [DC IC]
          <==> false                                     [prop-taut]
          <==> (x in null)                               [NC]]))


#define diff-remove-theorem := (forall A x . A - x = A \ singleton x)
#(mark 'A)
# START_LOAD
# datatype-cases diff-remove-theorem {
#   null => pick-any x
#             (!set-identity-intro-direct
#                pick-any y
#                  (!chain [(y in null - x)
#                      <==> (y in null)
#                      <==> false
#                      <==> (y in null & ~ y in singleton x)
#                      <==> (y in null \ singleton x)]))
# | (A as (insert h t)) =>
#      pick-any x
```

```
#         (!set-identity-intro
#            (!subset-intro
#               pick-any y
#                 assume hyp := (y in A - x)
#                  (!two-cases
#                     assume case-1 := (x = h)
#                       let {y=/=x := (!chain [(y in A - x)
#                                              ==> (y in t - x)
#                                              ==> (y in t \ singleton x)
#                           ==> (y in t & ~ y in singleton x)
#                          ==> (y =/= x)])
# }
#END_LOAD

#(!induction* diff-remove-theorem)

conclude absorption-1 :=
  (forall x A . x in A <==> x ++ A = A)
  pick-any x A
    (!equiv
       assume hyp := (x in A)
          (!set-identity-intro-direct
             pick-any y
               (!chain [(y in x ++ A)
                  <==> (y = x | y in A)    [in-def]
                  <==> (y in A | y in A)   [hyp prop-taut]
                  <==> (y in A)            [prop-taut]]))
       assume (x ++ A = A)
         (!chain-> [true ==> (x in x ++ A) [in-lemma-1]
                         ==> (x in A)      [set-identity-characterization]]))

conclude subset-theorem-1 :=
  (forall A B . A subset B ==> A \/ B = B)
pick-any A B
  assume (A subset B)
    (!set-identity-intro-direct
      pick-any x
        (!chain [(x in A \/ B)
            <==> (x in A | x in B) [UC]
            <==> (x in B | x in B) [prop-taut SC]
            <==> (x in B)          [prop-taut]]))


conclude subset-theorem-2 :=
  (forall A B . A subset B ==> A /\ B = A)
  pick-any A B
    assume (A subset B)
      (!set-identity-intro-direct
        pick-any x
          (!chain [(x in A /\ B)
              <==> (x in A & x in B) [IC]
              <==> (x in A & x in A) [prop-taut SC]
              <==> (x in A)          [prop-taut]]))


conclude intersection-lemma-1 :=
  (forall A B x . x in B & x in A ==> A /\ B = (x ++ A) /\ B)
pick-any A B x
   assume hyp := (x in B &  x in A)
    (!set-identity-intro-direct
      pick-any y
        (!chain [(y in A /\ B)
            <==> (y in A & y in B)           [IC]
            <==> ((y = x | y in A) & y in B) [(y in A <==> y = x | y in A) <== (x in A) [in-lemma-4]]
            <==> ((y in x ++ A) & y in B)    [in-def]
            <==> (y in (x ++ A) /\ B)        [IC]]))

conclude intersection-lemma-2 :=
   (forall A B x . ~ x in A ==> ~ x in A /\ B)
pick-any A B x
```

```
  assume hyp := (~ x in A)
    (!by-contradiction (~ x in A /\ B)
       (!chain [(x in A /\ B)
            ==> (x in A)            [IC]
            ==> (x in A & ~ x in A) [augment]
            ==> false               [prop-taut]]))


conclude intersection-lemma-3 :=
   (forall A . A /\ A = A)
pick-any A
 (!set-identity-intro-direct
   pick-any x
     (!chain [(x in A /\ A)
         <==> (x in A & x in A)  [IC]
         <==> (x in A)           [prop-taut]]))

declare insert-in-all: (S) [S (Set (Set S))] -> (Set (Set S)) [[id lst->set]]

assert* insert-in-all-def :=
  [(x insert-in-all null = null)
   (x insert-in-all A ++ t = (x ++ A) ++ (x insert-in-all t))]

define in-all := insert-in-all

conclude insert-in-all-characterization :=
  (forall U s x . s in x in-all U <==> exists B . B in U & s = x ++ B)
by-induction insert-in-all-characterization {
  (U as null) => pick-any s x
                  (!equiv (!chain [(s in x in-all U)
                                ==> (s in null)                     [insert-in-all-def]
                                ==> false                           [NC]
                                ==> (exists B . B in U & s = x ++ B) [prop-taut]])
                          assume hyp := (exists B . B in U & s = x ++ B)
                            pick-witness B for hyp
                              (!chain-> [(B in U)
                                     ==> false    [NC]
                                     ==> (s in x in-all U)  [prop-taut]]))
| (U as (insert A more)) =>
   let {IH := (forall s x . s in x in-all more <==> exists B . B in more & s = x ++ B)}
    pick-any s x
    let {
         G := (exists B . B in U & s = x ++ B);
         L := conclude ((s = x ++ A | exists B . B in more & s = x ++ B) <==> G)
               (!equiv
                 assume hyp := (s = x ++ A | exists B . B in more & s = x ++ B)
                   (!cases hyp
                      assume (s = x ++ A)
                        (!chain-> [true ==> (A in U)               [in-lemma-1]
                                       ==> (s = x ++ A & A in U)  [augment]
                                       ==> (A in U & s = x ++ A)  [comm]
                                       ==> G                      [existence]])
                      (!chain [(exists B . B in more & s = x ++ B)
                            ==> (exists B . B in U & s = x ++ B)    [in-def]]))
                 assume hyp := (exists B . B in U & s = x ++ B)
                   let {goal := (s = x ++ A | exists B . B in more & s = x ++ B)}
                   pick-witness B for hyp
                      (!cases (!chain<- [(B = A | B in more) <== (B in U) [in-def]])
                        assume (B = A)
                          (!chain-> [(s = x ++ B) ==> (s = x ++ A) [(B = A)]
                                                  ==> goal         [alternate]])
                        assume (B in more)
                          (!chain-> [(B in more)
                                 ==> (B in more & s = x ++ B)            [augment]
                                 ==> (exists B . B in more & s = x ++ B) [existence]
                                 ==> goal                                [alternate]])))
      }
      (!chain [(s in x in-all U)
          <==> (s in (x ++ A) ++ (x in-all more)) [insert-in-all-def]
          <==> (s = x ++ A | s in x in-all more)  [in-def]
```

```
          <==> (s = x ++ A | exists B . B in more & s = x ++ B) [IH]
          <==> G                                              [L]
              ])
}

declare powerset: (S) [(Set S)] -> (Set (Set S)) [[lst->set]]

assert* powerset-def :=
  [(powerset null = singleton null)
   (powerset x ++ t = (powerset t) \/ (x insert-in-all (powerset t)))]

conclude powerset-characterization :=
  (forall A B . B in powerset A <==> B subset A)
by-induction powerset-characterization {
  (A as Set.null) =>
    pick-any B
      (!chain [(B in powerset A)
          <==> (B in singleton null)  [powerset-def]
          <==> (B = null)             [singleton-characterization]
          <==> (B subset null)        [subset-lemma-6]])
| (A as (Set.insert h t:(Set.Set 'S))) =>
    let {IH := (forall B . B in powerset t <==> B subset t)}
    pick-any B:(Set.Set 'S)
      let {e1 := (!chain [(B in powerset A)
                     <==> (B in (powerset t) \/ (h in-all powerset t))   [powerset-def]
                     <==> (B in powerset t | B in h in-all powerset t)   [UC]
                     <==> (B subset t      | B in h in-all powerset t)   [IH]
                     <==> (B subset t | exists s . s in powerset t & B = h ++ s) [insert-in-all-characterization]
                     <==> (B subset t | exists s . s subset t & B = h ++ s) [IH]]);
           lemma := (!chain-> [true ==> (h in h ++ t) [in-lemma-1]]);
           p3 := (assume hyp := (B subset t | exists s . s subset t & B = h ++ s)
                   (!cases hyp
                     (!chain [(B subset t) ==> (B subset A) [subset-lemma-5]])
                       (assume ehyp := (exists s . s subset t & B = h ++ s)
                         pick-witness s for ehyp
                           (!subset-intro
                             pick-any x
                               assume (x in B)
                                 (!chain-> [(x in B) ==> (x in h ++ s)    [(B = h ++ s)]
                                                     ==> (x = h | x in s) [in-def]
                                                     ==> (x in h ++ t | x in s) [lemma]
                                                     ==> (x in A | x in t)      [SC]
                                                     ==> (x in A | x in A)    [in-def]
                                                     ==> (x in A)             [prop-taut]]))))));
            p4 := (assume (B subset A)
                    (!two-cases
                      assume case1 := (h in B)
                        (!chain-> [(B subset A)
                               ==> (B subset A & h in B) [augment]
                               ==> (exists s . s subset t & B = h ++ s)  [subset-lemma-3]
                               ==> (B subset t | exists s . s subset t & B = h ++ s)  [alternate]])
                      assume case2 := (~ h in B)
                        (!chain-> [case2 ==> (~ h in B & B subset A) [augment]
                                         ==> (B subset t)            [subset-lemma-4]
                                         ==> (B subset t | exists s . s subset t & B = h ++ s) [alternate]])));
            p3<=>p4 := (!equiv p3 p4)}
       (!equiv-tran e1 p3<=>p4)
}

define POSC := powerset-characterization

conclude ps-theorem-1 := (forall A . null in powerset A)
  pick-any A
   (!chain-> [true ==> (null subset A)      [subset-def]
                   ==> (null in powerset A) [POSC]])

conclude ps-theorem-2 := (forall A . A in powerset A)
  pick-any A
    (!chain-> [true ==> (A subset A)      [subset-reflexivity]
                    ==> (A in powerset A) [POSC]])
```

```
conclude ps-theorem-3 :=
  (forall A B . A subset B <==> powerset A subset powerset B)
pick-any A B
  (!equiv assume (A subset B)
            (!subset-intro
              pick-any C
                (!chain [(C in powerset A)
                     ==> (C subset A)              [POSC]
                     ==> (C subset B)              [subset-transitivity]
                     ==> (C in powerset B)         [POSC]]))
          assume (powerset A subset powerset B)
            (!chain-> [true ==> (A in powerset A)  [ps-theorem-2]
                            ==> (A in powerset B)  [SC]
                            ==> (A subset B)       [POSC]]))

conclude ps-theorem-4 :=
  (forall A B . powerset A /\ B = (powerset A) /\ (powerset B))
pick-any A B
  (!set-identity-intro-direct
    pick-any C
      (!chain
        [(C in powerset A /\ B)
    <==> (C subset A /\ B)                   [POSC]
    <==> (C subset A & C subset B)           [intersection-subset-theorem']
    <==> (C in powerset A & C in powerset B) [POSC]
    <==> (C in (powerset A) /\ (powerset B)) [IC]]))

conclude ps-theorem-5 :=
  (forall A B . (powerset A) \/ (powerset B) subset powerset A \/ B)
pick-any A B
  (!subset-intro
    pick-any C
      (!chain [(C in (powerset A) \/ (powerset B))
           ==> (C in powerset A | C in powerset B)  [UC]
           ==> (C subset A | C subset B)            [POSC]
           ==> (C subset A \/ B)                    [union-subset-theorem]
           ==> (C in powerset A \/ B)               [POSC]]))

declare paired-with: (S, T) [S (Set T)] -> (Set (Pair S T))
                                            [130 [id lst->set]]

assert* paired-with-def :=
  [(_ paired-with null = null)
   (x paired-with h ++ t = x @ h ++ (x paired-with t))]

(eval 3 paired-with [2 8])

conclude paired-with-characterization :=
   (forall B x y a . x @ y in a paired-with B <==> x = a & y in B)
  by-induction paired-with-characterization {
    null => pick-any x y a
                (!chain [(x @ y in a paired-with null)
                    <==> (x @ y in null)     [paired-with-def]
                    <==> false               [null-characterization]
                    <==> (x = a & false)     [prop-taut]
                    <==> (x = a & y in null) [null-characterization]])
  | (B as (insert h t)) =>
      pick-any x y a
        let {IH := (forall x y a . x @ y in a paired-with t <==> x = a & y in t)}
          (!chain
            [(x @ y in a paired-with h ++ t)
        <==> (x @ y in a @ h ++ (a paired-with t))      [paired-with-def]
        <==> (x @ y = a @ h | x @ y in a paired-with t) [in-def]
        <==> (x = a & y = h | x @ y in a paired-with t) [pair-axioms]
        <==> (x = a & y = h | x = a & y in t)           [IH]
        <==> (x = a & (y = h | y in t))                 [prop-taut]
        <==> (x = a & y in B)                           [in-def]])
   }
```

```
conclude paired-with-lemma-1 :=
  (forall A  x . x paired-with A = null ==> A = null)
datatype-cases paired-with-lemma-1 {
  null => pick-any x
            (!chain [(x paired-with null = null)
                 ==> (null  = null)              [paired-with-def]])
| (insert h t) =>
   pick-any x
    (!chain
      [(x paired-with h ++ t = null)
   ==> (x @ h ++ (x paired-with t) = null)                [paired-with-def]
   ==> (forall z . ~ z in x @ h ++ (x paired-with t))     [NC-2]
   ==> (forall z . ~ (z = x @ h | z in x paired-with t)) [in-def]

   ==> (forall z . z =/= x @ h)                       [prop-taut]
   ==> (x @ h =/= x @ h)                              [(uspec with x @ h)]
   ==> (x @ h =/= x @ h & x @ h = x @ h)              [augment]
   ==> false                                          [prop-taut]
   ==> (h ++ t = null)                                [prop-taut]])
}

declare product: (S, T) [(Set S) (Set T)] -> (Set (Pair S T)) [150 [lst->set lst->set]]

define X := product

assert* product-def :=
  [(null X _ = null)
   (h ++ t X A = h paired-with A \/ t X A)]


(eval [1 2] X ['foo 'bar 'car])


conclude cartesian-product-characterization :=
  (forall A B a b . a @ b in A X B <==> a in A & b in B)
by-induction cartesian-product-characterization {
    null => pick-any B a b
             (!chain [(a @ b in null X B)
                 <==> (a @ b in null)        [product-def]
                 <==> false                  [null-characterization]
                 <==> (a in null & b in B)   [prop-taut null-characterization]])
  | (A as (insert h t)) =>
      let {IH := (forall B a b . a @ b in t X B <==> a in t & b in B)}
       pick-any B a b
         (!chain [(a @ b in h ++ t X B)
             <==> (a @ b in h paired-with B \/ t X B)           [product-def]
             <==> (a @ b in h paired-with B | a @ b in t X B)   [UC]
             <==> (a = h & b in B | a in t & b in B)            [paired-with-characterization IH]
             <==> ((a = h | a in t) & b in B)                   [prop-taut]
             <==> (a in A & b in B)                             [in-def]])
}

define CPC := cartesian-product-characterization

conclude cartesian-product-characterization-2 :=
  (forall x A B . x in A X B <==> exists a b . x = a @ b & a in A & b in B)
pick-any x A B
  (!equiv
     assume hyp := (x in A X B)
       let {p := (!chain-> [true ==> (exists a b . x = a @ b) [pair-axioms]])}
         pick-witnesses a b for p x=a@b
           (!chain-> [x=a@b ==> (a @ b in A X B) [hyp]
                            ==> (a in A & b in B) [CPC]
                            ==> (x=a@b & a in A & b in B) [augment]
                            ==> (exists a b . x = a @ b & a in A & b in B) [existence]])
     assume hyp := (exists a b . x = a @ b & a in A & b in B)
      pick-witnesses a b for hyp spec-premise
         (!chain-> [spec-premise
                ==> (a in A & b in B)   [prop-taut]
                ==> (a @ b in A X B)    [CPC]
```

```
                ==> (x in A X B)        [(x = a @ b)]]))

define CPC-2 := cartesian-product-characterization-2

define taut := (method (p q) (!sprove-from q [p]))

conclude product-theorem-1 :=
  (forall A B . A X B = null ==> A = null | B = null)
datatype-cases product-theorem-1 {
  null => pick-any B
            (!chain [(null X B = null)
                 ==> (null = null)      [product-def]
                 ==> (null = null | B = null) [alternate]])
 | (A as (insert h t)) =>
     pick-any B
       (!chain [(h ++ t X B = null)
            ==> (h paired-with B \/ t X B = null)       [product-def]
            ==> (h paired-with B = null & t X B = null) [union-null-theorem]
            ==> (B = null)                              [paired-with-lemma-1]
            ==> (h ++ t = null | B = null)                   [alternate]])
}

conclude product-theorem-2 :=
  (forall A B . A X B = null <==> A = null | B = null)
  pick-any A:(Set 'T1)  B:(Set 'T2)
     (!chain [(A X B = null)
         <==> (forall x . ~ x in A X B)                                 [NC-2]
         <==> (forall x . ~ exists a b . x = a @ b & a in A & b in B)   [CPC-2]
         <==> (forall x a b . a in A & b in B ==> x =/= a @ b)          [taut]
         <==> (forall a b . a in A & b in B ==> forall x . x =/= a @ b) [taut]
         <==> (forall a b . a in A & b in B ==> false)                  [taut]
         <==> (forall a b . ~ a in A | ~ b in B)                        [taut]
         <==> ((forall a . ~ a in A) | (forall b . ~ b in B))           [taut]
         <==> (A = null | B = null)                                     [NC-2]])

conclude product-theorem-3 :=
  (forall A B . non-empty A & non-empty B ==> A X B = B X A <==> A = B)
pick-any A:(Set 'S) B:(Set 'T)
  assume hyp := (non-empty A & non-empty B)
    let {p1 := (!chain-> [(non-empty A) ==> (exists a . a in A) [NC-3]]);
         p2 := (!chain-> [(non-empty B) ==> (exists b . b in B) [NC-3]]);
         M := method (S1 S2 c2) # assumes c2 in S2, S1 X S2 = S2 X S1,
                (!subset-intro  # and derives (S1 subset S2)
                  pick-any x
                    (!chain [(x in S1)
                         ==> (x in S1 & c2 in S2) [augment]
                         ==> (x @ c2 in S1 X S2)  [CPC]
                         ==> (x @ c2 in S2 X S1)  [SIC]
                         ==> (x in S2 & c2 in S1) [CPC]
                         ==> (x in S2)            [left-and]]))
         }
      pick-witness a for p1 # (a in A)
        pick-witness b for p2  # (b in B)
          (!equiv
             assume hyp := (A X B = B X A)
               (!set-identity-intro (!M A B b) (!M B A a))
             assume hyp := (A = B)
              (!chain-> [(A X A = A X A) ==> (A X B = B X A) [hyp]]))

conclude product-theorem-4 :=
  (forall A  B C . non-empty A &  A X B subset A X C ==> B subset C)
pick-any A B C
  assume hyp := (non-empty A & A X B subset A X C)
    pick-witness a for (!chain-> [hyp ==> (exists a . a in A) [NC-3]])
      (!subset-intro
         pick-any b
           (!chain [(b in B)
                ==> (a in A & b in B) [augment]
                ==> (a @ b in A X B)  [CPC]
                ==> (a @ b in A X C)  [SC]
```

```
                  ==> (a in A & b in C) [CPC]
                  ==> (b in C)          [right-and]]))

define pair-converter :=
  method (premise)
    match premise {
      (forall u:'S (forall v:'T body)) =>
        pick-any p:(Pair 'S 'T)
          let {E := (!chain-> [true ==> (exists ?x:'S ?y:'T .
                                          p = ?x @ ?y) [pair-axioms]])}
           pick-witnesses x y for E
             let {body' := (!uspec* premise [x y])}
               (!chain-> [body'
                    ==> (replace-term-in-sentence (x @ y) body' p)
                          [(p = x @ y)]])
    }

conclude product-theorem-5 :=
  (forall A B C . B subset C ==> A X B subset A X C)
pick-any A B C
 assume (B subset C)
   (!subset-intro
      (!pair-converter
          pick-any a b
            (!chain [(a @ b in A X B)
                 ==> (a in A & b in B)  [CPC]
                 ==> (a in A & b in C)  [SC]
                 ==> (a @ b in A X C)   [CPC]]))))

conclude product-theorem-6 :=
 (forall A B C . A X (B /\ C) = A X B /\ A X C)
pick-any A B C
  (!set-identity-intro-direct
    (!pair-converter
       pick-any x y
        (!chain [(x @ y in A X (B /\ C))
            <==> (x in A & y in B /\ C)                 [CPC]
            <==> (x in A & y in B & y in C)             [IC]
            <==> ((x in A & y in B) & (x in A & y in C)) [prop-taut]
            <==> (x @ y in A X B & x @ y in A X C)      [CPC]
            <==> (x @ y in A X B /\ A X C)              [IC]]))))

# Theorem 103:
conclude product-theorem-7 :=
 (forall A B C . A X (B \/ C) = A X B \/ A X C)
pick-any A B C
  (!set-identity-intro-direct
    (!pair-converter
       pick-any x y
        (!chain [(x @ y in A X (B \/ C))
            <==> (x in A & y in B \/ C)                 [CPC]
            <==> (x in A & (y in B | y in C))           [UC]
            <==> ((x in A & y in B) | (x in  A & y in C)) [prop-taut]
            <==> (x @ y in A X B | x @ y in A X C)     [CPC]
            <==> (x @ y in A X B \/ A X C)              [UC]]))))

# Theorem 104:
conclude product-theorem-8 :=
 (forall A B C . A X (B \ C) = A X B \ A X C)
pick-any A B C
  (!set-identity-intro-direct
    (!pair-converter
       pick-any x y
         (!chain [(x @ y in A X (B \ C))
             <==> (x in A & y in B \ C)                 [CPC]
             <==> (x in A & y in B & ~ y in C)           [DC]
             <==> ((x in A & y in B) & (~x in A | ~ y in C)) [prop-taut]
             <==> ((x in A & y in B) & ~ (x in A & y in C)) [prop-taut]
             <==> (x @ y in A X B & ~ x @ y in A X C)       [CPC]
             <==> (x @ y in A X B \ A X C)                  [DC]]))))
```

```
define [R R1 R2 R3 R4] :=
       [?R:(Set (Pair 'T14 'T15)) ?R1:(Set (Pair 'T16 'T17))
        ?R2:(Set (Pair 'T18 'T19)) ?R3:(Set (Pair 'T20 'T21))
        ?R4:(Set (Pair 'T22 'T23))]

#========================= RELATION DOMAINS AND RANGES

declare dom: (S, T) [(Set (Pair S T))] -> (Set S) [150 [lst->set]]

assert* dom-def :=
  [(dom null = null)
   (dom x @ _ ++ t = x ++ dom t)]

(eval dom [('a @ 1) ('b @ 2) ('c @ 98)])

declare range: (S, T) [(Set (Pair S T))] -> (Set T) [150 [lst->set]]

assert* range-def :=
  [(range null = null)
   (range _ @ y ++ t = y ++ range t)]

(eval range [('a @ 1) ('b @ 2) ('c @ 98)])

conclude in-dom-lemma-1 :=
  (forall R a x y . a = x ==> a in dom x @ y ++ R)
pick-any R a x y
  (!chain [(a = x) ==> (a in x ++ dom R) [in-def]
                   ==> (a in dom x @ y ++ R) [dom-def]])

conclude in-range-lemma-1 :=
  (forall R a x y . a = y ==> a in range x @ y ++ R)
pick-any R a x y
  (!chain [(a = y) ==> (a in y ++ range R) [in-def]
                   ==> (a in range x @ y ++ R) [range-def]])

conclude in-dom-lemma-2 :=
  (forall R x a b . x in dom R ==> x in dom a @ b ++ R)
pick-any R x a b
  (!chain   [(x in dom a @ b ++ R)
         <== (x in a ++ dom R)      [dom-def]
         <== (x in dom R)           [in-def]])

conclude in-range-lemma-2 :=
  (forall R y a b . y in range R ==> y in range a @ b ++ R)
pick-any R y a b
  (!chain   [(y in range a @ b ++ R)
         <== (y in b ++ range R)        [range-def]
         <== (y in range R)             [in-def]])


conclude dom-characterization :=
 (forall R x . x in dom R <==> exists y . x @ y in R)
by-induction dom-characterization {
  null =>  pick-any x
            (!chain [(x in dom null)
                <==> (x in null)                 [dom-def]
                <==> false                       [NC]
                <==> (exists y . false)          [taut]
                <==> (exists y . x @ y in null)  [NC]])

| (R as (insert (pair a:'S b) t)) =>
    let {IH := (forall x . x in dom t <==> exists y . x @ y in t)}
      pick-any x:'S
        let {p1 := assume hyp := (x in dom R)
                     (!cases (!chain<- [(x = a | x in dom t)
                                    <== (x in a ++ dom t) [in-def]
                                    <== hyp               [dom-def]])

                        assume case1 := (x = a)
```

```
                         (!chain-> [true ==> (a @ b in R) [in-lemma-1]
                                         ==> (x @ b in R) [case1]
                                         ==> (exists y . x @ y in R) [existence]])

                       assume case2 := (x in dom t)
                         (!chain-> [case2 ==> (exists y . x @ y in t) [IH]
                                          ==> (exists y . x @ y in R) [ in-def]]));
             p2 := (!chain [(exists y . x @ y in R)
                       ==> (exists y . x @ y = a @ b | x @ y in t) [in-def]
                       ==> (exists y . x = a | x @ y in t)          [pair-axioms]
                       ==> (exists y . x in dom R | x @ y in t)     [in-dom-lemma-1]
                       ==> (exists y . x in dom R | exists z . x @ z in t)     [in-dom-lemma-1 taut]
                       ==> (exists y . x in dom R | x in dom t)     [IH]
                       ==> (exists y . x in dom R | x in dom R)     [in-dom-lemma-2]
                       ==> (x in dom R)                             [taut]])
             }
        (!equiv p1 p2)
}

define DOMC := dom-characterization

conclude range-characterization :=
 (forall R y . y in range R <==> exists x . x @ y in R)
by-induction range-characterization {
  null =>  pick-any y
            (!chain [(y in range null)
                <==> (y in null)                 [range-def]
                <==> false                       [NC]
                <==> (exists y . false)          [taut]
                <==> (exists x . x @ y in null)  [NC]])

| (R as (insert (pair a b:'T) t)) =>
    let {IH := (forall y . y in range t <==> exists x . x @ y in t)}
      pick-any y:'T
        let {p1 := assume hyp := (y in range R)
                    (!cases (!chain<- [(y = b | y in range t)
                                   <== (y in b ++ range t) [in-def]
                                   <== hyp                 [range-def]])

                       assume case1 := (y = b)
                         (!chain-> [true ==> (a @ b in R) [in-lemma-1]
                                         ==> (a @ y in R) [case1]
                                         ==> (exists x . x @ y in R) [existence]])

                       assume case2 := (y in range t)
                         (!chain-> [case2 ==> (exists x . x @ y in t) [IH]
                                          ==> (exists x . x @ y in R) [ in-def]]));
             p2 := (!chain [(exists x . x @ y in R)
                       ==> (exists x . x @ y = a @ b | x @ y in t) [in-def]
                       ==> (exists x . y = b | x @ y in t)          [pair-axioms]
                       ==> (exists x . y in range R | x @ y in t)     [in-range-lemma-1]

                       ==> (exists x . y in range R | exists z . z @ y in t)     [in-range-lemma-1 taut]

                       ==> (exists x . y in range R | y in range t)     [IH]

                       ==> (exists x . y in range R | y in range R)     [in-range-lemma-2]
                       ==> (y in range R)                             [taut]])
             }
        (!equiv p1 p2)
}

define RANC := range-characterization

conclude dom-theorem-1 :=
  (forall R1 R2 . dom (R1 \/ R2) = dom R1 \/ dom R2)
pick-any R1 R2
  (!set-identity-intro-direct
    pick-any x
      (!chain
```

```
        [(x in dom (R1 \/ R2))
    <==> (exists y . x @ y in R1 \/ R2)                          [DOMC]
    <==> (exists y . x @ y in R1 | x @ y in R2)                  [UC]
    <==> ((exists y . x @ y in R1) | (exists y . x @ y in R2)) [taut]
    <==> (x in dom R1 | x in dom R2)                             [DOMC]
    <==> (x in dom R1 \/ dom R2)                                 [UC]]))


conclude range-theorem-1 :=
  (forall R1 R2 . range (R1 \/ R2) = range R1 \/ range R2)
pick-any R1 R2
  (!set-identity-intro-direct
     pick-any y
       (!chain [(y in range (R1 \/ R2))
           <==> (exists x . x @ y in R1 \/ R2)                          [RANC]
           <==> (exists x . x @ y in R1 | x @ y in R2)                  [UC]
           <==> ((exists x . x @ y in R1) | (exists x . x @ y in R2)) [taut]
           <==> (y in range R1 | y in range R2)                         [RANC]
           <==> (y in range R1 \/ range R2)                             [UC]]))


conclude dom-theorem-2 :=
  (forall R1 R2 . dom (R1 /\ R2) subset dom R1 /\ dom R2)
pick-any R1 R2
 (!subset-intro
    pick-any x
      (!chain [(x in dom (R1 /\ R2))
           ==> (exists y . x @ y in R1 /\ R2)   [DOMC]
           ==> (exists y . x @ y in R1 & x @ y in R2) [IC]
           ==> ((exists y . x @ y in R1) & (exists y . x @ y in R2)) [taut]
           ==> (x in dom R1 & x in dom R2)   [DOMC]
           ==> (x in dom R1 /\ dom R2)       [IC]]))

(falsify (forall R1 R2 . dom (R1 /\ R2) = dom R1 /\ dom R2) 10)

conclude range-theorem-2 :=
  (forall R1 R2 . range (R1 /\ R2) subset range R1 /\ range R2)
pick-any R1 R2
 (!subset-intro
    pick-any y
      (!chain [(y in range (R1 /\ R2))
           ==> (exists x . x @ y in R1 /\ R2)   [RANC]
           ==> (exists x . x @ y in R1 & x @ y in R2) [IC]
           ==> ((exists x . x @ y in R1) & (exists x . x @ y in R2)) [taut]
           ==> (y in range R1 & y in range R2)   [RANC]
           ==> (y in range R1 /\ range R2)       [IC]]))


conclude dom-theorem-3 :=
  (forall R1 R2 . dom R1 \ dom R2 subset dom (R1 \ R2))
 pick-any R1 R2
   (!subset-intro
      pick-any x
        assume hyp := (x in dom R1 \ dom R2)
          let {lemma := (!chain-> [hyp ==> (x in dom R1 & ~ x in dom R2) [DC]])}
           pick-witness w for (!chain-> [lemma ==> (x in dom R1) [left-and]
                                               ==> (exists y . x @ y in R1) [DOMC]])
             (!chain-> [lemma ==> (~ x in dom R2)                 [right-and]
                          ==> (~ exists y . x @ y in R2) [DOMC]
                          ==> (forall y . ~ x @ y in R2) [qn]
                          ==> (~ x @ w in R2)   [(uspec with w)]
                          ==> (x @ w in R1 & ~ x @ w in R2) [augment]
                          ==> (exists y . x @ y in R1 & ~ x @ y in R2) [existence]
                          ==> (exists y . x @ y in R1 \ R2)             [DC]
                          ==>  (x in dom (R1 \ R2))            [DOMC]]))


conclude range-theorem-3 :=
  (forall R1 R2 . range R1 \ range R2 subset range (R1 \ R2))
 pick-any R1 R2
```

```
   (!subset-intro
      pick-any y
        assume hyp := (y in range R1 \ range R2)
          let {lemma := (!chain-> [hyp ==> (y in range R1 & ~ y in range R2) [DC]])}
           pick-witness w for (!chain-> [lemma ==> (y in range R1) [left-and]
                                               ==> (exists x . x @ y in R1) [RANC]])
             (!chain-> [lemma ==> (~ y in range R2)                 [right-and]
                        ==> (~ exists x . x @ y in R2) [RANC]
                        ==> (forall x . ~ x @ y in R2) [qn]
                        ==> (~ w @ y in R2)  [(uspec with w)]
                        ==> (w @ y in R1 & ~ w @ y in R2) [augment]
                        ==> (exists x . x @ y in R1 & ~ x @ y in R2) [existence]
                        ==> (exists x . x @ y in R1 \ R2)            [DC]
                        ==>  (y in range (R1 \ R2))            [RANC]]))



declare conv: (S, T) [(Set (Pair S T))] -> (Set (Pair T S)) [210 [lst->set]]
define -- := conv

assert* conv-def :=
  [(-- null = null)
   (-- x @ y ++ t = y @ x ++ -- t)]


define pair-lemma-1 := Pair.pair-theorem-2

conclude converse-characterization :=
  (forall R x y . x @ y in -- R <==> y @ x in R)
by-induction converse-characterization {
  null => pick-any x y
           (!chain [(x @ y in -- null)
               <==> (x @ y in null)     [conv-def]
               <==> false               [NC]
               <==> (y @ x in null)     [NC]])

| (R as (insert (pair a b) t)) =>
    let {
         IH := (forall x y . x @ y in -- t <==> y @ x in t)}
      pick-any x y
        (!chain [(x @ y in -- R)
            <==> (x @ y in b @ a ++ -- t)           [conv-def]
            <==> (x @ y = b @ a | x @ y in -- t)    [in-def]
            <==> (y @ x = a @ b | x @ y in -- t)    [pair-lemma-1]
            <==> (y @ x = a @ b | y @ x in t)      [IH]
            <==> (y @ x in R)                      [in-def]])
      }


conclude converse-theorem-1 :=
  (forall R . -- -- R = R)
by-induction converse-theorem-1 {
  null => (!chain [(-- -- null) = (-- null) [conv-def]
                                = null     [conv-def]])
| (R as (insert (pair x y) t)) =>
    let {IH := (-- -- t = t)}
     (!chain [(-- -- x @ y ++ t)
            = (-- (y @ x ++ -- t))  [conv-def]
            = (x @ y ++ -- -- t)    [conv-def]
            = (x @ y ++ t)        [IH]])
}

conclude converse-theorem-2 :=
  (forall R1 R2 . -- (R1 /\ R2) = -- R1 /\ -- R2)
 pick-any R1 R2
   (!set-identity-intro-direct
     (!pair-converter
        pick-any x y
          (!chain [(x @ y in -- (R1 /\ R2))
              <==> (y @ x in R1 /\ R2)             [converse-characterization]
```

```
              <==> (y @ x in R1 & y @ x in R2)     [IC]
              <==> (x @ y in -- R1 & x @ y in -- R2) [converse-characterization]
              <==> (x @ y in -- R1 /\ -- R2)         [IC]]))))


conclude converse-theorem-3 :=
  (forall R1 R2 . -- (R1 \/ R2) = -- R1 \/ -- R2)
 pick-any R1 R2
   (!set-identity-intro-direct
     (!pair-converter
        pick-any x y
          (!chain [(x @ y in -- (R1 \/ R2))
              <==> (y @ x in R1 \/ R2)             [converse-characterization]
              <==> (y @ x in R1 | y @ x in R2)     [UC]
              <==> (x @ y in -- R1 | x @ y in -- R2) [converse-characterization]
              <==> (x @ y in -- R1 \/ -- R2)         [UC]]))))


conclude converse-theorem-4 :=
  (forall R1 R2 . -- (R1 \ R2) = -- R1 \ -- R2)
 pick-any R1 R2
   (!set-identity-intro-direct
     (!pair-converter
        pick-any x y
          (!chain [(x @ y in -- (R1 \ R2))
              <==> (y @ x in R1 \ R2)                [converse-characterization]
              <==> (y @ x in R1 & ~ y @ x in R2)     [DC]
              <==> (x @ y in -- R1 & ~ x @ y in -- R2) [converse-characterization]
              <==> (x @ y in -- R1 \ -- R2)            [DC]]))))


declare composed-with: (S1, S2, S3) [(Pair S1 S2) (Set (Pair S2 S3))] -> (Set (Pair S1 S3)) [200 [id lst->set]]

assert* composed-with-def :=
  [(_ composed-with null = null)
   (x @ y composed-with z @ w ++ t = x @ w ++ (x @ y composed-with t) <== y = z)
   (x @ y composed-with z @ w ++ t = x @ y composed-with t <== y =/= z)]



(eval 1 @ 2 composed-with [(2 @ 5) (7 @ 8) (2 @ 3)])
(eval 1 @ 2 composed-with [(7 @ 8) (9 @ 10)])
(eval 1 @ 2 composed-with [])

conclude composed-with-characterization :=
  (forall R x y z w . w @ z in x @ y composed-with R <==> w = x & y @ z in R)
by-induction composed-with-characterization {
  (R as null) => pick-any x y z w
                   (!chain [(w @ z in x @ y composed-with null)
                       <==> (w @ z in null)   [composed-with-def]
                       <==> false             [NC]
                       <==> (w = x & y @ z in null)   [prop-taut NC]])

| (R as (insert (pair a b) t)) =>
    pick-any x y z w
      let {IH := (forall x y z w . w @ z in x @ y composed-with t <==> w = x & y @ z in t)}
       (!two-cases
         assume case1 := (y = a)
           (!chain [(w @ z in x @ y composed-with a @ b ++ t)
               <==> (w @ z in x @ b ++ (x @ y composed-with t)) [composed-with-def]
               <==> (w @ z = x @ b | w @ z in x @ y composed-with t) [in-def]
               <==> (w @ z = x @ b | (w = x & y @ z in t))                   [IH]
               <==> (w = x & z = b | w = x & y @ z in t)        [pair-axioms]
               <==> (w = x & y = a & z = b | w = x & y @ z in t) [augment]
               <==> (w = x & y @ z = a @ b | w = x & y @ z in t) [pair-axioms]
               <==> (w = x & (y @ z = a @ b | y @ z in t)) [prop-taut]
               <==> (w = x & y @ z in R)                 [in-def]])
         assume case2 := (y =/= a)
           (!iff-comm
             (!chain [(w = x & y @ z in R)
```

```
                   <==> (w = x & (y @ z = a @ b | y @ z in t))          [in-def]
                   <==> (w = x & (y = a & z = b | y @ z in t))          [pair-axioms]
                   <==> (w = x & (case2 & y = a & z = b | y @ z in t)) [augment]
                   <==> (w = x & (false | y @ z in t))                  [prop-taut]
                   <==> (w = x & y @ z in t)                            [prop-taut]
                   <==> (w @ z in x @ y composed-with t) [IH]
                   <==> (w @ z in x @ y composed-with R) [composed-with-def]]))))
}

conclude composed-with-characterization' :=
  (forall R x y z . x @ z in x @ y composed-with R <==> y @ z in R)
pick-any R x y z
  (!chain [(x @ z in x @ y composed-with R)
      <==> (x = x & y @ z in R)  [composed-with-characterization]
      <==> (y @ z in R)          [augment]])


declare o: (S1, S2, S3) [(Set (Pair S1 S2)) (Set (Pair S2 S3))] -> (Set (Pair S1 S3)) [200 [lst->set lst->set]]

assert* o-def :=
 [(null o _ = null)
  (x @ y ++ t o R = x @ y composed-with R \/ t o R)]

(eval [('nyc @ 'boston) ('houston @ 'dallas) ('austin @ 'dc)] o

      [('boston @ 'montreal) ('dallas @ 'chicago) ('dc @ 'nyc)] o
      [('chicago @ 'seattle) ('montreal @ 'london)])

let {R1 := [('nyc @ 'boston) ('austin @ 'dc)];
     R2 := [('boston @ 'montreal) ('dc @ 'chicago) ('chicago @ 'seattle)]}
  (eval R1 o R2)

conclude o-characterization :=
  (forall R1 R2 x z . x @ z in R1 o R2 <==> exists y . x @ y in R1 & y @ z in R2)
by-induction o-characterization {
  (R1 as null) => pick-any R2 x z
                    (!chain [(x @ z in R1 o R2)
                         <==> (x @ z in null)   [o-def]
                         <==> false             [NC]
                         <==> (exists y . false & y @ z in R2) [(method (p q) (!force q))]
                         <==> (exists y . x @ y in null & y @ z in R2) [NC (method (p q) (!force q))]])
| (R1 as (insert (pair a b) t)) =>
    pick-any R2 x z
      let {IH := (forall R2 x z . x @ z in t o R2 <==> exists y . x @ y in t & y @ z in R2)}
         let {dir1 := assume hyp := (x @ z in R1 o R2)
                        (!cases (!chain-> [hyp
                                      ==> (x @ z in a @ b composed-with R2 \/ t o R2)           [o-def]
                                      ==> (x @ z in a @ b composed-with R2 | x @ z in t o R2)   [UC]
                                      ==> (x @ z in a @ b composed-with R2 | exists y . x @ y in t & y @ z in R2) [IH]
                                      ==> (x @ z in a @ b composed-with R2 | exists y . x @ y in R1 & y @ z in R2) [in
                                      ==> (x = a & b @ z in R2  | exists y . x @ y in R1 & y @ z in R2) [composed-with
                          assume case1 := (x = a & b @ z in R2)
                            (!chain-> [true ==> (a @ b in R1) [in-lemma-1]
                                            ==> (x @ b in R1) [case1]
                                            ==> (x @ b in R1 & b @ z in R2) [augment]
                                            ==> (exists y . x @ y in R1 & y @ z in R2) [taut]])
                          assume case2 := (exists y . x @ y in R1 & y @ z in R2)
                            (!claim case2));
               dir2 := assume hyp := (exists y . x @ y in R1 & y @ z in R2)
                        pick-witness y for hyp
                          (!cases (!chain-> [(x @ y in R1)
                                           ==> (x @ y = a @ b | x @ y in t) [in-def]])
                            assume case1 := (x @ y = a @ b)
                              let {_ := (!chain-> [case1 ==> (x = a) [pair-axioms]]);
                                   _ := (!chain-> [case1 ==> (y = b) [pair-axioms]])}
                              (!chain-> [(x = a)
                                      ==> (x = a & y @ z in R2) [augment]
                                      ==> (x = a & b @ z in R2) [(y = b)]
                                      ==> (x @ z in a @ b composed-with R2) [composed-with-characterization]
                                      ==> (x @ z in a @ b composed-with R2 \/ t o R2) [UC]
```

```
                                        ==> (x @ z in R1 o R2) [o-def]])
                              assume case2 := (x @ y in t)
                               (!chain-> [case2
                                      ==> (x @ y in t & y @ z in R2) [augment]
                                      ==> (exists y . x @ y in t & y @ z in R2) [existence]
                                      ==> (x @ z in t o R2) [IH]
                                      ==> (x @ z in a @ b composed-with R2 | x @ z in t o R2) [prop-taut]
                                      ==> (x @ z in a @ b composed-with R2 \/ t o R2) [UC]
                                      ==> (x @ z in R1 o R2)                          [o-def]]))
                 }
         (!equiv dir1 dir2)
}



conclude compose-theorem-1 :=
  (forall R1 R2 . dom R1 o R2 subset dom R1)
pick-any R1 R2
  (!subset-intro
     pick-any x
       (!chain [(x in dom R1 o R2)
            ==> (exists y . x @ y in R1 o R2)                      [dom-characterization]
            ==> (exists y . exists z . x @ z in R1 & z @ y in R2)   [o-characterization]
            ==> (exists y . exists z . x @ z in R1)                 [taut]
            ==> (exists y . x in dom R1)                            [dom-characterization]
            ==> (x in dom R1)                                       [taut]]))

conclude compose-theorem-2 :=
  (forall R1 R2 R3 R4 . R1 subset R2 & R3 subset R4 ==> R1 o R3 subset R2 o R4)
pick-any R1:(Set (Pair 'S 'T)) R2:(Set (Pair 'S 'T))
         R3:(Set (Pair 'T 'U)) R4:(Set (Pair 'T 'U))
  assume hyp :=  (R1 subset R2 & R3 subset R4)
    (!subset-intro
       (!pair-converter
          pick-any x y
            (!chain [(x @ y in R1 o R3)
                 ==> (exists z . x @ z in R1 & z @ y in R3)  [o-characterization]
                 ==> (exists z . x @ z in R2 & z @ y in R3)  [SC]
                 ==> (exists z . x @ z in R2 & z @ y in R4)  [SC]
                 ==> (x @ y in R2 o R4)                      [o-characterization]])))

conclude compose-theorem-3 :=
  (forall R1 R2 R3  . R1 o (R2 \/ R3) = R1 o R2 \/ R1 o R3)
pick-any R1 R2 R3
  (!set-identity-intro-direct
     (!pair-converter
        pick-any x y
          (!chain [(x @ y in R1 o (R2 \/ R3))
              <==> (exists z . x @ z in R1 & z @ y in R2 \/ R3) [o-characterization]
              <==> (exists z . x @ z in R1 & (z @ y in R2 | z @ y in R3)) [UC]
              <==> (exists z . x @ z in R1 & z @ y in R2 | x @ z in R1 & z @ y in R3) [prop-taut]
              <==> ((exists z . x @ z in R1 & z @ y in R2) | (exists z . x @ z in R1 & z @ y in R3)) [taut]
              <==> (x @ y in R1 o R2 |  x @ y in R1 o R3) [o-characterization]
              <==> (x @ y in R1 o R2 \/ R1 o R3) [UC]])))

conclude compose-theorem-4 :=
  (forall R1 R2 R3  . R1 o (R2 /\ R3) subset R1 o R2 /\ R1 o R3)
pick-any R1 R2 R3
  (!subset-intro
     (!pair-converter
        pick-any x y
          (!chain [(x @ y in R1 o (R2 /\ R3))
               ==> (exists z . x @ z in R1 & z @ y in R2 /\ R3) [o-characterization]
               ==> (exists z . x @ z in R1 & (z @ y in R2 & z @ y in R3)) [IC]
               ==> (exists z . (x @ z in R1 & z @ y in R2) & (x @ z in R1 & z @ y in R3)) [prop-taut]
               ==> ((exists z . x @ z in R1 & z @ y in R2) & (exists z . x @ z in R1 & z @ y in R3)) [taut]
               ==> (x @ y in R1 o R2 & x @ y in R1 o R3) [o-characterization]
               ==> (x @ y in R1 o R2 /\ R1 o R3)         [IC]])))
```

```
conclude compose-theorem-5 :=
  (forall R1 R2 R3 . R1 o R2 \ R1 o R3 subset R1 o (R2 \ R3))
pick-any R1 R2 R3
  (!subset-intro
     (!pair-converter
        pick-any x y
          (!chain [(x @ y in R1 o R2 \ R1 o R3)
               ==> (x @ y in R1 o R2 & ~ x @ y in R1 o R3) [DC]
               ==> ((exists z . x @ z in R1 & z @ y in R2) & ~ (exists z . x @ z in R1  & z @ y in R3)) [o-characteriz
               ==> (exists z . x @ z in R1 & z @ y in R2 & ~ z @ y in R3)  [taut]
               ==> (exists z . x @ z in R1 & z @ y in R2 \ R3)  [DC]
               ==> (x @ y in R1 o (R2 \ R3))  [o-characterization]]))))

conclude composition-assoc :=
  (forall R1 R2 R3 . R1 o R2 o R3 = (R1 o R2) o R3)
pick-any R1 R2 R3
  (!set-identity-intro-direct
     (!pair-converter
        pick-any x y
          (!chain [(x @ y in R1 o R2 o R3)
              <==> (exists z . x @ z in R1 & z @ y in R2 o R3)                    [o-characterization]
              <==> (exists z . x @ z in R1 & exists w . z @ w in R2 & w @ y in R3) [o-characterization]
              <==> (exists w z . x @ z in R1 & z @ w in R2 & w @ y in R3)          [taut]
              <==> (exists w . (exists z . x @ z in R1 & z @ w in R2)  & w @ y in R3)          [taut]
              <==> (exists w . x @ w in R1 o R2  & w @ y in R3)          [o-characterization]
              <==> (x @ y in (R1 o R2) o R3)                             [o-characterization]]))))

 conclude compose-theorem-6 :=
   (forall R1 R2 . -- (R1 o R2) = -- R2 o -- R1)
 pick-any R1 R2
   (!set-identity-intro-direct
     (!pair-converter
       pick-any x y
          (!chain [(x @ y in -- (R1 o R2))
              <==> (y @ x in R1 o R2)       [converse-characterization]
              <==> (exists z . y @ z in R1 & z @ x in R2) [o-characterization]
              <==> (exists z . z @ y in -- R1 & x @ z in -- R2) [converse-characterization]
              <==> (exists z . x @ z in -- R2 & z @ y in -- R1) [prop-taut]
              <==> (x @ y in -- R2 o -- R1)                  [o-characterization]]))))


declare restrict1: (S, T) [(Set (Pair S T)) S] -> (Set (Pair S T)) [200 [lst->set id]]

assert* restrict1-def :=
[(null restrict1 _ = null)
 (x @ y ++ t restrict1 z = x @ y ++ (t restrict1 z) <== x = z)
 (x @ y ++ t restrict1 z = t restrict1 z <== x =/= z)]

(eval [(1 @ 'foo) (2 @ 'b) (1 @ 'bar)] restrict1 1)

define restrict1-characterization :=
 (forall R x y a . x @ y in R restrict1 a <==> x @ y in R & x = a)

(define ^1 restrict1)

conclude restrict1-lemma :=
 (forall R x y a . x @ y in R & x = a ==> x @ y in R ^1 a)
by-induction restrict1-lemma {
 (R as null) => pick-any x y a
                 (!chain [(x @ y in R & x = a)
                      ==> (x @ y in R)          [left-and]
                      ==> false                 [NC]
                      ==> (x @ y in R ^1 a)     [prop-taut]])
| (R as (insert (pair x' y') t)) =>
     let {IH := (forall x y a . x @ y in t & x = a ==> x @ y in t ^1 a)}
       pick-any x y a
         assume hyp := (x @ y in R & x = a)
           (!two-cases
              assume case1 := (x' = a)
```

```
                (!chain-> [hyp
                       ==> ((x @ y = x' @ y' | x @ y in t) & x = a) [in-def]
                       ==> (x @ y = x' @ y' & x = a | x @ y in t & x = a) [prop-taut]
                       ==> (x @ y in x' @ y' ++ (t ^1 a) & x = a | x @ y in t & x = a) [in-def]
                       ==> (x @ y in R ^1 a & x = a | x @ y in t & x = a) [restrict1-def]
                       ==> (x @ y in R ^1 a & x = a | x @ y in t ^1 a)    [IH]
                       ==> (x @ y in R ^1 a & x = a | x @ y in x' @ y' ++ (t ^1 a)) [in-def]
                       ==> (x @ y in R ^1 a & x = a | x @ y in R ^1 a)    [restrict1-def]
                       ==> (x @ y in R ^1 a)  [prop-taut]])
              assume case2 := (x' =/= a)
                (!cases (!chain-> [hyp
                               ==> ((x @ y = x' @ y' | x @ y in t) & x = a) [in-def]
                               ==> ((x = x' & y = y' | x @ y in t) & x = a) [pair-axioms]
                               ==> (x = x' & y = y' & x = a | x @ y in t & x = a) [prop-taut]])
                    assume hyp1 := (x = x' & y = y' & x = a)
                       let {_ := (!absurd (!chain-> [hyp1 ==> (x = a)
                                                         ==> (x' = a)])
                                          case2)}
                         (!from-false (x @ y in R ^1 a))
                    assume hyp2 := (x @ y in t & x = a)
                      (!chain-> [hyp2 ==> (x @ y in t ^1 a)   [IH]
                                      ==> (x @ y in R ^1 a)   [restrict1-def]]))))
}

by-induction restrict1-characterization {
  (R as null) => pick-any x y a
                   (!chain [(x @ y in R ^1 a)
                       <==> (x @ y in null)         [restrict1-def]
                       <==> false                   [NC]
                       <==> (false & x = a)         [prop-taut]
                       <==> (x @ y in R & x = a)    [NC]])
| (R as (insert (pair x' y') t)) =>
    pick-any x y a
      let {IH := (forall x y a . x @ y in t ^1 a <==> x @ y in t & x = a);
           goal := (x @ y in R ^1 a <==> x @ y in R & x = a);
           dir1 := assume hyp := (x @ y in R ^1 a)
                    (!two-cases
                       assume case1 := (x' = a)
                         (!cases (!chain-> [hyp
                                        ==> (x @ y in x' @ y' ++ (t ^1 a))       [restrict1-def]
                                        ==> (x @ y = x' @ y' | x @ y in t ^1 a)  [in-def]])
                            assume hyp1a := (x @ y = x' @ y')
                              (!both (!chain-> [hyp1a ==> (x @ y in R) [in-def]])
                                     (!chain-> [hyp1a ==> (x = x') [pair-axioms]
                                                      ==> (x = a)           [case1]]))
                            (!chain [(x @ y in t ^1 a) ==> (x @ y in t & x = a) [IH]
                                                       ==> (x @ y in R & x = a) [in-def]]))
                       assume case2 := (x' =/= a)
                         (!chain-> [hyp ==> (x @ y in t ^1 a)    [restrict1-def]
                                        ==> (x @ y in t & x = a) [IH]
                                        ==> (x @ y in R & x = a) [in-def]]));
           dir2 := (!chain [(x @ y in R & x = a) ==> (x @ y in R ^1 a) [restrict1-lemma]])}
       (!equiv dir1 dir2)
}


declare restrict: (S, T) [(Set (Pair S T)) (Set S)] -> (Set (Pair S T)) [200 [lst->set lst->set]]

define ^ := restrict
assert* restrict-def :=
[(R restrict null = null)
 (R restrict h ++ t = R restrict1 h \/ R restrict t)]

(eval [(1 @ 'foo) (2 @ 'b) (3 @ 'c) (4 @ 'd) (1 @ 'bar)] ^ [1 2])

conclude restrict-characterization :=
 (forall A R x y . x @ y in R restrict A <==> x @ y in R & x in A)
by-induction restrict-characterization {
  (A as null) => pick-any R x y
                   (!chain [(x @ y in R restrict A)
```

```
                       <==> (x @ y in null)          [restrict-def]
                       <==> false                    [NC]
                       <==> (x @ y in R & false)     [prop-taut]
                       <==> (x @ y in R & x in A)   [NC]])
| (A as (insert h t)) =>
    let {IH := (forall R x y . x @ y in R restrict t <==> x @ y in R & x in t)}
      pick-any R x y
        (!chain [(x @ y in R restrict A)
            <==> (x @ y in R ^1 h \/ R restrict t)         [restrict-def]
            <==> (x @ y in R ^1 h | x @ y in R restrict t) [UC]
            <==> ((x @ y in R & x = h) | x @ y in R restrict t) [restrict1-characterization]
            <==> ((x @ y in R & x = h) | x @ y in R & x in t)   [IH]
            <==> ((x @ y in R) & (x = h | x in t))              [prop-taut]
            <==> (x @ y in R & x in A)                          [in-def]])
}

conclude restriction-theorem-1 :=
 (forall R A B . A subset B ==> R ^ A subset R ^ B)
pick-any R A B
  assume (A subset B)
    (!subset-intro
       (!pair-converter
          pick-any x y
           (!chain [(x @ y in R ^ A)
                ==> (x @ y in R & x in A) [restrict-characterization]
                ==> (x @ y in R & x in B) [SC]
                ==> (x @ y in R ^ B)      [restrict-characterization]])))


conclude restriction-theorem-2 :=
 (forall R A B . R ^ (A /\ B) = R ^ A /\ R ^ B)
pick-any R A B
  (!set-identity-intro-direct
    (!pair-converter
      pick-any x y
        (!chain [(x @ y in R ^ (A /\ B))
            <==> (x @ y in R & x in A /\ B)                       [restrict-characterization]
            <==> (x @ y in R & x in A & x in B)                   [IC]
            <==> ((x @ y in R & x in A) & (x @ y in R & x in B)) [prop-taut]
            <==> (x @ y in R ^ A & x @ y in R ^ B)                [restrict-characterization]
            <==> (x @ y in R ^ A /\ R ^ B)                        [IC]])))


conclude restriction-theorem-3 :=
 (forall R A B . R ^ (A \/ B) = R ^ A \/ R ^ B)
pick-any R A B
  (!set-identity-intro-direct
    (!pair-converter
      pick-any x y
        (!chain [(x @ y in R ^ (A \/ B))
            <==> (x @ y in R & x in A \/ B)                       [restrict-characterization]
            <==> (x @ y in R & (x in A | x in B))                 [UC]
            <==> ((x @ y in R & x in A) | (x @ y in R & x in B)) [prop-taut]
            <==> (x @ y in R ^ A | x @ y in R ^ B)                [restrict-characterization]
            <==> (x @ y in R ^ A \/ R ^ B)                        [UC]])))


conclude restriction-theorem-4 :=
 (forall R A B . R ^ (A \ B) = R ^ A \ R ^ B)
pick-any R A B
  (!set-identity-intro-direct
    (!pair-converter
      pick-any x y
        (!chain [(x @ y in R ^ (A \ B))
            <==> (x @ y in R & x in A \ B)                       [restrict-characterization]
            <==> (x @ y in R & (x in A & ~ x in B))                   [DC]
            <==> ((x @ y in R & x in A) & ~ (x @ y in R & x in B)) [prop-taut]
            <==> (x @ y in R ^ A & ~ x @ y in R ^ B)                [restrict-characterization]
            <==> (x @ y in R ^ A \ R ^ B)                        [DC]])))
```

```
conclude restriction-theorem-5 :=
 (forall R1 R2 A . (R1 o R2) ^ A = (R1 ^ A) o R2)
pick-any R1 R2 A
  (!set-identity-intro-direct
    (!pair-converter
      pick-any x y
        (!chain [(x @ y in (R1 o R2) ^ A)
            <==> (x @ y in R1 o R2 & x in A)                      [restrict-characterization]
            <==> ((exists z . x @ z in R1 & z @ y in R2) & x in A)   [o-characterization]
            <==> (exists z . x @ z in R1 & z @ y in R2 & x in A)     [taut]
            <==> (exists z . (x @ z in R1 & x in A) & z @ y in R2)   [prop-taut]
            <==> (exists z . x @ z in R1 ^ A & z @ y in R2)          [restrict-characterization]
            <==> (x @ y in (R1 ^ A) o R2)                           [o-characterization]]))))


declare image: (S, T) [(Set (Pair S T)) (Set S)] -> (Set T) [** 200 [lst->set lst->set]]

#define ** := image

assert* image-def :=  [(R ** A = range R ^ A)]

(eval [(1 @ 'a) (2 @ 'b) (3 @ 'c)] ** [1 3])

conclude image-characterization :=
  (forall R A y . y in R ** A <==> exists x . x @ y in R & x in A)
pick-any R A y
  (!chain [(y in R ** A)
      <==> (y in range R ^ A)   [image-def]
      <==> (exists x . x @ y in R ^ A) [range-characterization]
      <==> (exists x . x @ y in R  & x in A) [restrict-characterization]])

conclude image-lemma :=
  (forall R A x y . x @ y in R & x in A ==> y in R ** A)
pick-any R A x y
  (!chain [(x @ y in R & x in A)
       ==> (exists x . x @ y in R & x in A) [existence]
       ==> (y in R ** A)                    [image-characterization]])

conclude image-theorem-1 :=
  (forall R A B . R ** (A \/ B) = R ** A \/ R ** B)
pick-any R A B
  (!set-identity-intro-direct
      pick-any y
        (!chain [(y in R ** (A \/ B))
           <==> (exists x . x @ y in R & x in A \/ B)  [image-characterization]
           <==> (exists x . x @ y in R & (x in A | x in B))  [UC]
           <==> (exists x . (x @ y in R & x in A) | (x @ y in R & x in B)) [prop-taut]
           <==> ((exists x . x @ y in R & x in A) | (exists x . x @ y in R & x in B)) [taut]
           <==> (y in R ** A | y in R ** B)  [image-characterization]
           <==> (y in R ** A \/ R ** B)      [UC]]))


conclude image-theorem-2 :=
  (forall R A B . R ** (A /\ B) subset R ** A /\ R ** B)
pick-any R A B
  (!subset-intro
      pick-any y
        (!chain [(y in R ** (A /\ B))
           ==> (exists x . x @ y in R & x in A /\ B)  [image-characterization]
           ==> (exists x . x @ y in R & x in A & x in B)  [IC]
           ==> (exists x . (x @ y in R & x in A)  & (x @ y in R & x in B)) [prop-taut]
           ==> ((exists x . x @ y in R & x in A) & (exists x . x @ y in R & x in B)) [taut]
           ==> (y in R ** A & y in R ** B) [image-characterization]
           ==> (y in R ** A /\ R ** B) [IC]]))


conclude image-theorem-3 :=
  (forall R A B . R ** A \ R ** B subset R ** (A \ B))
```

```
pick-any R A B
  (!subset-intro
     pick-any y
       (!chain [(y in R ** A \ R ** B)
           ==> (y in R ** A & ~ y in R ** B)  [DC]
           ==> ((exists x . x @ y in R & x in A) & ~ (exists x . x @ y in R & x in B))   [image-characterization]
           ==> ((exists x . x @ y in R & x in A) & (forall x . x @ y in R ==> ~ x in B))  [taut]
           ==> (exists x . x @ y in R & x in A & ~ x in B) [taut]
           ==> (exists x . x @ y in R & x in A \ B)        [DC]
           ==> (y in R ** (A \ B))                         [image-characterization]]))

conclude image-theorem-4 :=
  (forall R A B . A subset B ==> R ** A subset R ** B)
pick-any R A B
  assume hyp := (A subset B)
    (!subset-intro
      pick-any y
        (!chain [(y in R ** A)
            ==> (exists x . x @ y in R & x in A) [image-characterization]
            ==> (exists x . x @ y in R & x in B) [SC]
            ==> (y in R ** B)                    [image-characterization]]))

conclude image-theorem-5 :=
  (forall R A . R ** A = null <==> dom R /\ A = null)
pick-any R A
  (!chain [(R ** A = null)
     <==>  (forall y . ~ y in R ** A)  [null-characterization-2]
     <==> (forall y . ~ exists x . x @ y in R & x in A) [image-characterization]
     <==> (forall x . ~ exists y . x @ y in R & x in A) [taut]
     <==> (forall x . ~ ((exists y . x @ y in R) & x in A)) [taut]
     <==> (forall x . ~ (x in dom R & x in A)) [dom-characterization]
     <==> (forall x . ~ (x in dom R /\ A)) [IC]
     <==> (dom R /\ A = null)  [null-characterization-2]])


conclude image-theorem-6 :=
  (forall R A . dom R /\ A subset -- R ** R ** A)
pick-any R A
  (!subset-intro
    pick-any x
      (!chain [(x in dom R /\ A)
          ==> (x in dom R & x in A) [IC]
          ==> ((exists y . x @ y in R) & x in A) [dom-characterization]
          ==> (exists y . x @ y in R & x @ y in R & x in A)   [taut]
          ==> (exists y . x @ y in R & y in R ** A)   [image-lemma]
          ==> (exists y . y @ x in -- R & y in R ** A)   [converse-characterization]
          ==> (x in -- R ** R ** A)                     [image-characterization]]))

(falsify (forall R A . dom R /\ A = -- R ** R ** A) 20)

conclude image-theorem-7 :=
  (forall R A B . (R ** A) /\ B subset R ** (A /\ -- R ** B))
pick-any R A B
  (!subset-intro
    pick-any y
      (!chain [(y in (R ** A) /\ B)
          ==> (y in R ** A & y in B) [IC]
          ==> ((exists x . x @ y in R & x in A) & y in B) [image-characterization]
          ==> (exists x . x @ y in R & x in A & y in B)    [taut]
          ==> (exists x .   y @ x in -- R & x in A & x @ y in R & y in B)    [converse-characterization augment]
          ==> (exists x .   (y @ x in -- R & y in B) & x in A & x @ y in R)    [prop-taut]
          ==> (exists x .   x in -- R ** B & x in A & x @ y in R)    [image-lemma]
          ==> (exists x .   x @ y in R & (x in A & x in -- R ** B))    [prop-taut]
          ==> (exists x .   x @ y in R & x in A /\ -- R ** B)    [IC]
          ==> (y in R ** (A /\ -- R ** B))            [image-characterization]]))


define lemma := (close t /\ (x insert-in-all t) = null)
define lemma2 := (close (forall y . y in t ==> ~ x in y) ==> t /\ (x insert-in-all t) = null)
```

```
declare card: (S) [(Set S)] -> N [[lst->set]]

define S := N.S

assert* card-def :=
  [(card null = zero)
   (card h ++ t = card t <== h in t)
   (card h ++ t = S card t <== ~ h in t)]

transform-output eval [nat->int]

(eval card [1 2 3] \/ [4 7 8])

define [< <=] := [N.< N.<=]

overload + N.+

define card-theorem-1 :=
  (card singleton _ = S zero)

conclude card-theorem-2 :=
  (forall A x . ~ x in A ==> card A < card x ++ A)
pick-any A x
  assume hyp := (~ x in A)
     (!chain-> [true ==> (card A < S card A)     [N.Less.<S]
                     ==> (card A < card x ++ A) [card-def]])

conclude minus-card-theorem :=
 (forall A x . x in A ==> card A = N.S card A - x)
by-induction minus-card-theorem {
  (A as null:(Set.Set 'S)) =>
     pick-any x
       (!chain [(x in A)
            ==> false     [NC]
            ==> (card A = N.S card A - x) [prop-taut]])
| (A as (insert h:'S t:(Set.Set 'S))) =>
    let {IH := (forall x . x in t ==> card t = N.S card (t - x))}
      pick-any x:'S
        assume hyp := (x in A)
          (!two-cases
             assume case1 := (x = h)
               (!two-cases
                 assume (h in t)
                   let {_ := (!chain-> [(h in t) ==> (x in t) [case1]])}
                    (!chain [(card A)
                           = (card t)              [card-def]
                           = (N.S (card t - x))    [IH]
                           = (N.S (card A - x))    [remove-def]])
                 assume (~ h in t)
                   let {_ := (!chain-> [(~ h in t) ==> (~ x in t) [case1]])}
                     (!combine-equations
                       (!chain [(card A) = (N.S card t)])
                       (!chain [(N.S card (A - x))
                             = (N.S card (t - x))
                             = (N.S card t)])))
                 assume case2 := (x =/= h)
                   let {_ := (!chain-> [(x in A)
                                    ==> (x = h | x in t)  [in-def]
                                    ==> (x in t)          [(dsyl with case2)]])}
                     (!two-cases
                       assume (h in t)
                         let {_ := (!chain-> [(h in t)
                                          ==> (h in t & x =/= h) [augment]
                                          ==> (h in t - x)       [remove-corollary-3]])}
                           (!chain [(card A)
                                  = (card t)                  [card-def]
                                  = (N.S card (t - x))        [IH]
                                  = (N.S card h ++ (t - x))   [card-def]
                                  = (N.S card (A - x))        [remove-def]])
                       assume (~ h in t)
```

```
                        let {_ := (!chain-> [(~ h in t) ==> (~ h in t - x) [remove-corollary-4]])}
                          (!chain-> [(card t) = (N.S card t - x) [IH]
                                 ==> (N.S card t = N.S N.S card t - x)
                                 ==> (card A     = N.S N.S card t - x)    [card-def]
                                 ==> (card A     = N.S card h ++ (t - x)) [card-def]
                                 ==> (card A     = N.S card A - x)        [remove-def]])))
}

define subset-card-theorem :=
  (forall A B . A subset B ==> card A <= card B)


by-induction subset-card-theorem {
  null => pick-any B:(Set.Set 'S)
           assume hyp := (null subset B)
               (!chain-> [true ==> (zero <= card B)   [N.Less=.zero<=]
                               ==> (card null:(Set.Set 'S) <= card B) [card-def]])
| (A as (insert h:'S t:(Set.Set 'S))) =>
   let {IH := (forall B . t subset B ==> card t <= card B)}
    pick-any B:(Set.Set 'S)
      assume hyp := (A subset B)
        (!two-cases
           assume case1 := (in h t)
             (!chain-> [hyp ==> (t subset B)          [subset-lemma-2]
                            ==> (card t <= card B)    [IH]
                            ==> (card A <= card B)    [card-def]])
           assume case2 := (~ in h t)
             let {t-sub-B := (!chain-> [hyp ==> (t subset B)          [subset-lemma-2]]);
                  _ := (!chain-> [true
                             ==> (in h A) [in-lemma-1]
                             ==> (in h B) [SC]])}
               (!chain-> [t-sub-B ==> (t subset B & case2)  [augment]
                                 ==> (t subset B - h)     [remove-corollary-5]
                                 ==> (card t <= card B - h) [IH]
                                 ==> (S card t <= S card B - h)
                                 ==> (S card t <= card B)     [minus-card-theorem]
                                 ==> (card A <= card B)       [card-def]]))
}

conclude proper-subset-card-theorem :=
  (forall A B . A proper-subset B ==> card A < card B)
pick-any A B
  assume hyp := (A proper-subset B)
    pick-witness x for (!chain-> [hyp ==> (A subset B & exists x . x in B & ~ x in A) [PSC]
                                      ==> (exists x . x in B & ~ x in A)              [right-and]])
      let {L1 := (!chain-> [hyp ==> (A subset B)       [PSC]
                                ==> (x ++ A subset B) [subset-lemma-1]
                                ==> (card x ++ A <= card B) [subset-card-theorem]]);
           L2 := (!chain-> [(~ x in A) ==> (card A < card x ++ A) [card-theorem-2]])}
        (!chain-> [L1 ==> (L1 & L2) [augment]
                      ==> (card A < card B)   [N.Less=.transitive1]])

conclude intersection-card-theorem-1 :=
  (forall A B . card A /\ B <= card A)
pick-any A B
  (!chain-> [true ==> (A /\ B subset A)       [intersection-subset-theorem]
                  ==> (card A /\ B <= card A) [subset-card-theorem]])

conclude intersection-card-theorem-2 :=
  (forall A B . card A /\ B <= card B)
pick-any A B
  (!chain-> [true ==> (A /\ B subset B)       [intersection-subset-theorem-2]
                  ==> (card A /\ B <= card B) [subset-card-theorem]])

conclude intersection-card-theorem-3 :=
  (forall A B x . ~ x in A & x in B ==> card (x ++ A) /\ B = N.S card A /\ B)
pick-any A B x
  assume hyp := (~ x in A & x in B)
    let {_ := (!chain-> [(~ x in A) ==> (~ x in A /\ B) [intersection-lemma-2]])}
      (!chain [(card (x ++ A) /\ B)
```

```
              = (card x ++ (A /\ B))   [intersection-def]
              = (S card A /\ B)        [card-def]])

# by-induction card-lemma-1 {
#   (A as (insert h t)) =>
#     let {_ := (mark 'A)}
#      (!vpf (forall B x . ~ x in A & x in B ==> card (x ++ A) /\ B = N.S card A /\ B) (ab))
# | (A as Set.null) => let {_ := (mark 'B)} (!dhalt)
# }

define card-lemma-2 :=
  (forall A B . card A \/ B = ((card A) + (card B)) N.- (card A /\ B))

overload - N.-

conclude num-lemma :=
  (forall x y z . (x + y) - z = (S x + y) - S z)
pick-any x:N y:N z:N
   (!chain-> [((S x +  y) -  S z)
            = (S (x + y) - S z)        [N.Plus.left-nonzero]
            = ((x + y) - z)            [N.Minus.axioms]
          ==> ((x + y) - z = (S x + y) - S z) [sym]])


conclude lemma-p1 :=
  (forall A B x . ~ x in A and x in B ==> card (x ++ A) /\ B = S card A /\ B)
pick-any A B x
  assume hyp := (~ x in A & x in B)
    let {_ := (!chain-> [(~ x in A)
                     ==> (~ x in A /\ B) [intersection-lemma-2]])}
      (!chain [(card x ++ A /\ B)
              = (card x ++ (A /\ B)) [intersection-def]
              = (S card A /\ B)      [card-def]])


conclude lemma-p2 :=
  (forall A B x . ~ x in A & ~ x in B ==> A /\ B = (x ++ A) /\ B)
pick-any A B x
  assume hyp := (~ x in A & ~ x in B)
    (!set-identity-intro-direct
      pick-any y
        (!equiv assume hyp1 := (y in A /\ B)
                  let {L1 := (!chain-> [hyp1 ==> (y in A) [IC]
                                             ==> (y = x | y in A) [alternate]
                                             ==> (y in x ++ A)    [in-def]]);
                       L2 := (!chain-> [hyp1 ==> (y in B) [IC]])}
                   (!chain-> [L1
                           ==> (L1 & L2)            [augment]
                           ==> (y in (x ++ A) /\ B) [IC]])
                assume hyp2 := (y in (x ++ A) /\ B)
                  let {L1 := (!chain-> [hyp2 ==> (y in B)      [IC]]);
                       L2 := (!by-contradiction (y =/= x)
                               assume (y = x)
                                 (!chain-> [(y in B) ==> (x in B)            [(y = x)]
                                                     ==> (x in B & ~ x in B) [augment]
                                                     ==> false               [prop-taut]]))}
                    (!chain-> [hyp2 ==> (y in x ++ A)     [IC]
                                    ==> (y = x | y in A)  [in-def]
                                    ==> ((y = x | y in A) & y =/= x) [augment]
                                    ==> (((y = x) & (y =/= x)) | (y in A & y =/= x)) [prop-taut]
                                    ==> (false | y in A & y =/= x)                   [prop-taut]
                                    ==> (y in A)                                     [prop-taut]
                                    ==> (y in A & y in B)                            [augment]
                                    ==> (y in A /\ B)                                [IC]])))

 # by-induction card-lemma-2 {
 #  null => (!vpf   (forall B . card null \/ B = (card null) + (card B) N.- card null /\ B) (ab))
 # | (A as (insert h t)) =>
 #    let {_ := (mark 'A)}
 #       (!vpf (forall B . card A \/ B = (card A) + (card B) N.- card A /\ B) (ab))
```

```

 # }

#(falsify card-lemma-1 10)

conclude union-lemma-2 :=
  (forall A B x . x ++ (A \/ B) = A \/ x ++ B)
pick-any A B x
  (!chain [(x ++ (A \/ B))
        = (x ++ (B \/ A)) [union-commutes]
        = ((x ++ B) \/ A) [union-def]
        = (A \/ (x ++ B)) [union-commutes]])

conclude union-subset-lemma-1 := (forall A B . A subset A \/ B)
  pick-any A B
    (!subset-intro
       pick-any x
         (!chain [(x in A) ==> (x in A \/ B) [UC]]))

conclude union-subset-lemma-2 := (forall A B . B subset A \/ B)
  pick-any A B
    (!subset-intro
       pick-any x
         (!chain [(x in B) ==> (x in A \/ B) [UC]]))

conclude leq-lemma-1 := (forall x y . x <= x + y)
  pick-any x y
    (!by-contradiction (x <= x + y)
       let {-y<zero := (!chain-> [true ==> (~ y < zero) [N.Less.not-zero]])}
       (!chain [(~ x <= x + y)
            ==> (x + y < x)          [N.Less=.trichotomy1]
            ==> (x + y < x + zero)   [N.Plus.right-zero]
            ==> (y + x < zero + x)   [N.Plus.commutative]
            ==> (y < zero)           [N.Less.Plus-cancellation]
            ==> (y < zero & -y<zero) [augment]
            ==> false                [prop-taut]]))

conclude leq-lemma :=
  (forall x y z . x <= y ==> x <= y + z)
pick-any x y z
   assume hyp := (x <= y)
     (!chain-> [true ==> (y <= y + z)          [leq-lemma-1]
                     ==> (x <= y & y <= y + z) [augment]
                     ==> (x <= y + z)          [N.Less=.transitive]])

conclude minus-lemma :=
           (forall x y . y <= x ==> S (x - y) = (S x) - y)
pick-any x:N y:N
 assume (y <= x)
  let {_ := (!chain-> [(y <= x) ==> (y <= S x) [N.Less=.S2]])}
  (!chain-> [(S x) = (S x)
         ==> (S ((x - y) + y) = S x)          [N.Minus.Plus-Cancel]
         ==> (S (x - y) + y = S x)            [N.Plus.left-nonzero]
         ==> (S (x - y) + y = (S x - y) + y)  [N.Minus.Plus-Cancel]
         ==> (S (x - y) = S x - y)            [N.Plus.=-cancellation]])

conclude union-card :=
  (forall A B . card A \/ B = ((card A) + (card B)) - card A /\ B)
by-induction union-card {
  null => pick-any B
           let {ns := null:(Set.Set 'S)}
             (!chain [(card ns \/ B)
                         = (card B)                                [union-def]
                         = ((card B) - zero)                       [N.Minus.axioms]
                         = ((card B) - (card ns))                   [card-def]
                         = ((card B) - card ns /\ B)                [intersection-def]
                         = ((zero + card B) - card ns /\ B)         [N.Plus.left-zero]
                         = (((card ns) + (card B)) - card ns /\ B)   [card-def]])
| (A as (insert h t:(Set.Set 'S))) =>
    let {IH := (forall B . card t \/ B = ((card t) + (card B)) - card t /\ B)}
```

```
    pick-any B:(Set.Set 'S)
      (!two-cases
        assume case1 := (h in t)
          let {_ := (!chain-> [(h in t) ==> (h in t \/ B) [UC]]);
               L1 := (!chain [(card A \/ B)
                            = (card h ++ (t \/ B))                     [union-def]
                            = (card t \/ B)                            [card-def]
                            = (((card t) + (card B)) - (card t /\ B)) [IH]
                            = (((card A) + (card B)) - (card t /\ B)) [card-def]])}
            (!two-cases
              assume (h in B)
               let {_ := (!both (h in B) (h in t))}
                 (!chain [(card A \/ B)
                        = (((card A) + (card B)) - (card t /\ B))   [L1]
                        = (((card A) + (card B)) - (card A /\ B))   [intersection-lemma-1]])
              assume (~ h in B)
                 (!chain [(card A \/ B)
                        = (((card A) + (card B)) - (card t /\ B))   [L1]
                        = (((card A) + (card B)) - (card A /\ B))   [intersection-def]]))
        assume case2 := (~ h in t)
          (!two-cases
            assume (h in B)
              let {_ := (!chain-> [(h in B) ==> (h in t \/ B) [UC]]);
                   _ := (!chain-> [(~ h in t) ==> (~ h in t /\ B) [IC]])}
                (!chain [(card A \/ B)
                       = (card h ++ (t \/ B))                    [union-def]
                       = (card t \/ B)                           [card-def]
                       = (((card t) + (card B)) - (card t /\ B)) [IH]
                       = (((S card t) + (card B)) - (S (card t /\ B))) [num-lemma]
                       = (((card A) + (card B)) - (S (card t /\ B))) [card-def]
                       = (((card A) + (card B)) - (S (card t /\ B))) [card-def]
                       = (((card A) + (card B)) - (card h ++ (t /\ B))) [card-def]
                       = (((card A) + (card B)) - (card A /\ B))    [intersection-def]])
            assume (~ h in B)
              let {_ := (!chain-> [(~ h in t)
                                ==> (~ h in t & ~ h in B) [augment]
                                ==> (~ (h in t | h in B)) [dm]
                                ==>  (~ h in t \/ B)      [UC]]);
                   _ := (!chain-> [true
                                ==> (card t /\ B <= card t)                [intersection-card-theorem-1]
                                ==> (card t /\ B  <= (card t) + (card B)) [leq-lemma]])}
                (!chain [(card A \/ B)
                       = (card h ++ (t \/ B))   [union-def]
                       = (S card t \/ B)        [card-def]
                       = (S (((card t) + card B) - card t /\ B)) [IH]
                       = ((S  ((card t) + (card B))) - (card t /\ B)) [minus-lemma]
                       = ((S  ((card t) + (card B))) - (card A /\ B)) [lemma-p2]
                       = (((S card t) + card B) - (card A /\ B)) [N.Plus.left-nonzero]
                       = (((card A) + card B) - (card A /\ B)) [card-def]]))))
}


conclude diff-card-lemma :=
  (forall A B . card A = (card A \ B) + (card A /\ B))
pick-any A B
  (!chain-> [true ==> (A = (A \ B) \/ (A /\ B)) [diff-theorem-12]
                  ==> (card A = card (A \ B) \/ (A /\ B))
                  ==> (card A = ((card A \ B) + (card A /\ B)) - (card (A \ B) /\ (A /\ B))) [union-card]
                  ==> (card A = ((card A \ B) + (card A /\ B)) - (card null))                [diff-theorem-13]
                  ==> (card A = ((card A \ B) + (card A /\ B)) - zero)                [card-def]
                  ==> (card A = (card A \ B) + card A /\ B)                  [N.Minus.axioms]])

conclude diff-card-theorem :=
  (forall A B . card A \ B =  (card A) - card A /\ B)
pick-any A B
  (!chain-> [true ==> (card A = (card A \ B) + card A /\ B) [diff-card-lemma]
                  ==> ((card A \ B) + card A /\ B = card A) [sym]
                  ==> (card A \ B = (card A) - card A /\ B) [N.Minus.Plus-Minus-properties]])

declare fun: (S, T) [(Set (Pair S T))] -> Boolean [210 [lst->set]]
```

```
assert* fun-def :=
  [(fun null)
   (fun x @ y ++ t = fun t <== ~ x in dom t | t ** singleton x = singleton y)
   (~ fun x @ y ++ t <== ~ (~ x in dom t | t ** singleton x = singleton y))]


(eval fun [(1 @ 'a) (2 @ 'b)])

(eval fun [(1 @ 'a) (2 @ 'b) (1 @ 'c)])

(eval fun [(1 @ 'a) (2 @ 'b) (3 @ 'c) (2 @ 'd)])

(eval fun [(1 @ 'a) (2 @ 'b) (3 @ 'c) (8 @ 'd)])

(eval fun [])

(eval fun [(1 @ 'a)])

(eval fun [(1 @ 'a) (1 @ 'a)])

} # close Set

EOF
()

(load "sets")
```