# lib/basic/fmaps.ath

```
load "sets"

load "strong-induction"

module FMap {

define succ := (string->symbol "S")
define < := N.<

define [A B C] := [?A:(Set.Set 'S1) ?B:(Set.Set 'S2) ?C:(Set.Set 'S3)]

structure (Map S T) := empty-map
                     | (update (Pair S T) (Map S T))

assert (structure-axioms "Map")
define Pair := pair

define (alist->fmap-general L preprocessor) :=
  match L {
    [] => empty-map
  | (list-of (|| [x --> n] [x n]) rest) =>
         (update (pair (preprocessor x) (preprocessor n)) (alist->fmap-general rest preprocessor))
  | _ => L
  }

define (alist->fmap L) := (alist->fmap-general L id)

define (fmap->alist-general m preprocessor) :=
  match m {
    empty-map => []
  | (update (pair k v) rest) => (add [(preprocessor k) --> (preprocessor v)]
                                     (fmap->alist-general rest preprocessor))
  | _ => m
  }

define (fmap->alist m) := (fmap->alist-general m id)

define map-induction :=
 method (goal premises)
   match goal {
     (forall (some-var x) (some-sentence body)) =>
        let {property := lambda (m) (replace-var x m body)}
          by-induction goal {
            empty-map => (!vpf (property empty-map) premises)
          | (update p m) =>
               let {goal := (replace-var x (update p m) body);
                    IH :=  (property m)}
                 (!vpf goal (add IH premises))
           }
    }

define map-induction' :=
  method (goal)
    (!map-induction goal (ab))

define (alist->pair inner-1 inner-2) :=
  lambda (L)
    match L  {
      [a b] =>      ((inner-1 a) @ (inner-2 b))
    | [a --> b] => ((inner-1 a) @ (inner-2 b))
    | _ => L
    }

expand-input update [(alist->pair id id) alist->fmap]

define :: := Cons
```

```
define [null ++ in subset proper-subset \/ /\ \ - card] :=
       [Set.null Set.++ Set.in Set.subset Set.proper-subset
        Set.\/ Set./\ Set.\ Set.- Set.card]

overload ++ update
#set-precedence ++ 210



define [key key1 key2 k k' k1 k2] := [?key ?key1 ?key2 ?k ?k' ?k1 ?k2]
define [val val1 val2 v v' v1 v2 x x1 x2 y y1 y2] :=
       [?val ?val1 ?val2 ?v ?v' ?v1 ?v2 ?x ?x1 ?x2 ?y ?y1 ?y2]
define [m m' m1 m2 m3 rest rest1] := [?m:(Map 'S1 'S2) ?m':(Map 'S1 'S2) ?m1:(Map 'S3 'S4)
                                      ?m2:(Map 'S5 'S6) ?m3:(Map 'S7 'S8) ?rest:(Map 'S9 'S10) ?rest1:(Map 'S11 'S12)]
define [S S1 S2 S3] := [?S:(Set.Set 'S) ?S1:(Set.Set 'S1) ?S2:(Set.Set 'S2) ?S3:(Set.Set 'S3)]

define [L L1 L2 more more1] := [?L ?L1 ?L2 ?more ?more1]

declare apply: (S, T) [(Map S T) S] -> (Option T) [applied-to 110 [alist->fmap id]]

define at := applied-to

declare remove: (S, T) [(Map S T) S] -> (Map S T) [- 120 [alist->fmap id]]

left-assoc -

(define t1 (- ?x ?y))

define  (removed-from key map) := (remove map key)

#assert* remove-axioms :=
#  [(_ removed-from empty-map = empty-map)
#   (key removed-from [key _] ++ rest = key removed-from rest)
#   (key =/= x ==> x removed-from [key val] ++ rest = [key val] ++ (x removed-from rest))]

assert* remove-def :=
  [([] - _ = empty-map)
   ([key _] ++ rest - key = rest - key)
   (key =/= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]

(define t2 (- ?x null))
(define t3 (- ?x ?y))

#define (- map key) := (key removed-from map)



define M := [[1 --> 'a] [2 --> 'b] [1 --> 'c]]
define ide-map := [['a --> 1] ['b --> 2] ['c --> 3] ['a --> 99]]
define ide-map' := [['b --> 2] ['c --> 3] ['a --> 1]  ['a --> 99]]
define ide-map'' := [['b --> 2] ['c --> 3] ['a --> 1]  ['d --> 4] ['a --> 99]]

#(set-flag mlstyle-fundef "on")

assert* apply-axioms :=
   [([] at _ = NONE)
    ([key val] ++ _ at x = SOME val <== key = x)
    ([key _] ++ rest at x = rest at x <== key =/= x)]

#define applied-to := apply
## The following gives the result NONE:(Option 'T286327), but it should be NONE:(Option Int)
# (set-flag mlstyle-fundef "on")
#(apply` empty-map:(Map Int Int) 1) [FIXED]

(eval (empty-map:(Map Int Int) at  1))

(eval M applied-to 1)
(eval M applied-to 2)
(eval M applied-to 97)
(eval M - 1 applied-to 2)
```

```
(eval M - 1 applied-to 1)

conclude apply-lemma-1 :=
  (forall key val rest x .
     [key val] ++ rest at x = NONE ==> rest at x = NONE)
   pick-any key val rest x
    let {m := ([key val] ++ rest);
         hyp := (m at x = NONE);
         goal := (rest at x = NONE)}
      assume hyp
        (!two-cases
          (!chain [(key = x)
               ==> (m at x = SOME val)  [apply-axioms]
               ==> (m at x =/= NONE)    [option-results]
               ==> (hyp & ~hyp)         [augment]
               ==> goal                 [prop-taut]])
          (!chain [(key =/= x)
               ==> (m at x = rest at x) [apply-axioms]
               ==>  (NONE = rest at x)  [hyp]
               ==> goal                 [sym]]))



conclude apply-lemma-2 :=
  (forall k v rest x .
     [k v] ++ rest applied-to x =/= NONE <==> k = x | rest applied-to x =/= NONE)
pick-any k v rest x
  (!two-cases
     assume case-1 := (k = x)
       (!equiv assume hyp := ([k v] ++ rest applied-to x =/= NONE)
                  (!chain-> [(k = x) ==> (k = x | rest applied-to x =/= NONE) [alternate]])
               assume (k = x | rest applied-to x =/= NONE)
                  (!chain-> [([k v] ++ rest applied-to x)
                           = ([x v] ++ rest applied-to x) [(k = x)]
                           = (SOME v)              [apply-axioms]
                         ==> ([k v] ++ rest applied-to x =/= NONE) [option-results]]))
     assume case-2 := (k =/= x)
       (!equiv assume hyp := ([k v] ++ rest applied-to x =/= NONE)
                  (!chain-> [hyp
                         ==> (rest applied-to x =/= NONE)          [apply-axioms]
                         ==> (k = x | rest applied-to x =/= NONE)  [alternate]])
               assume C := (k = x | rest applied-to x =/= NONE)
                    (!cases C
                       assume (k = x)
                         (!from-complements ([k v] ++ rest applied-to x =/= NONE) (k = x) (k =/= x))
                      (!chain [(rest applied-to x =/= NONE) ==> ([k v] ++ rest applied-to x =/= NONE) [apply-lemma-1]]

conclude apply-lemma-3 :=
  (forall m k v1 v2 . m applied-to k = SOME v1 & m applied-to k = SOME v2 ==> v1 = v2)
 pick-any m k v1 v2
    assume hyp := (m applied-to k = SOME v1 & m applied-to k = SOME v2)
      (!chain-> [(SOME v1)
              = (m applied-to k)
              = (SOME v2)
            ==> (v1 = v2)  [option-results]])

conclude remove-correctness :=
  (forall m x . m - x applied-to x = NONE)
by-induction remove-correctness {
  (m as empty-map) =>
    pick-any x
      (!chain [([] - x applied-to x)
            = ([] applied-to x)                [remove-def]
            = NONE                     [apply-axioms]])
| (m as (update (pair key val) rest)) =>
   let {IH := (forall x . rest - x applied-to x = NONE)}
     pick-any x
       (!two-cases
          assume case1 := (key = x)
            (!chain [(m - x applied-to x)
```

```
                      = (m - key applied-to key)  [case1]
                      = (rest - x applied-to x)   [case1 remove-def]
                      = NONE                                 [IH]])
           assume case2 := (key =/= x)
             (!chain [(m - x applied-to x)
                      = ([key val] ++ (rest - x) applied-to x) [remove-def]
                      = (rest - x applied-to x)                [apply-axioms]
                      = NONE                                                 [IH]]))
}

define (RC2-M goal p1 p2) :=
  match [goal p1 p2] {
    [(~ (s = t)) (s = u) (~ (u = t))] =>
      (!by-contradiction goal
         assume (~ goal)
           (!chain-> [(~ goal)
                  ==> (s = t)           [dn]
                  ==> (u = t)           [(s = u)]
                  ==> (u = t & u =/= t) [augment]
                  ==> false             [prop-taut]]))
  }


conclude remove-correctness-2 :=
  (forall m x y . x =/= y ==> (m - x) at y = m at y)
by-induction remove-correctness-2 {
  (m as empty-map) =>
    pick-any x y
      assume hyp := (x =/= y)
        (!chain [((m - x) at y)
              = (m at y)                  [remove-def]])
| (m as (update (pair key val) rest)) =>
    let {IH := (forall x y . x =/= y ==> (rest - x) at y = rest at y)}
    pick-any x y
      assume hyp := (x =/= y)
        (!two-cases
           assume case1 := (key = x)
            #let {lemma := (!CongruenceClosure.cc (key =/= y) [case1 hyp])}
            let {lemma := (!RC2-M (key =/= y) case1 hyp)}
             (!chain [((m - x) at y)
                    = ((rest - x) at y)    [(key = x) remove-def]
                    = (rest at y)                     [IH]
                    = (m at y)                        [apply-axioms]])
           assume (key =/= x)
               (!two-cases
                  assume (key = y)
                    (!combine-equations
                       (!chain [((m - x) at y)
                              = (([key val] ++ (rest - x)) at y) [remove-def]
                              = (SOME val)                                          [apply-axioms]])
                       (!chain [(m at y)
                              = (SOME val)                                          [apply-axioms]]))
                  assume (key =/= y)
                    (!combine-equations
                       (!chain [((m - x) at y)
                              = (([key val] ++ (rest - x)) at y) [remove-def]
                           = ((rest - x) at y)              [apply-axioms]
                           = (rest at y)                                 [IH]])
                       (!chain [(m at y)
                             = (rest at y)                               [apply-axioms]])))))
}

declare map->set: (S, T) [(Map S T)] -> (Set.Set (Pair S T)) [[alist->fmap]]

assert* map->set-def :=
  [(map->set empty-map = null)
   (map->set [k v] ++ rest = (k @ v) ++ map->set rest - k)]

assert* map-identity := (m1 = m2 <==> map->set m1 = map->set m2)
```

```
(eval map->set ide-map)
(eval (alist->fmap ide-map) = (alist->fmap ide-map'))
(eval (alist->fmap ide-map) = (alist->fmap ide-map''))


conclude opair-lemma :=
  (forall x1 x2 y1 y2 A . x1 =/= x2 ==> x1 @ y1 in A <==> x1 @ y1 in x2 @ y2 ++ A)
pick-any x1:'S x2:'S y1:'T y2:'T A:(Set.Set (Pair 'S 'T))
  assume (x1 =/= x2)
    (!equiv (!chain [(x1 @ y1 in A)
                 ==> (x1 @ y1 in x2 @ y2 ++ A)              [Set.in-lemma-3]])
            (!chain [(x1 @ y1 in x2 @ y2 ++ A)
                   ==> (x1 @ y1 = x2 @ y2 | x1 @ y1 in A)   [Set.in-def]
                   ==> ((x1 = x2 & y1 = y2) | x1 @ y1 in A) [(datatype-axioms "Pair")]
                   ==> (x1 = x2 | x1 @ y1 in A)             [prop-taut]
                   ==> (x1 =/= x2 & (x1 = x2 | x1 @ y1 in A)) [augment]
                   ==> ((x1 =/= x2 & x1 = x2) | (x1 =/= x2 & x1 @ y1 in A)) [prop-taut]
                   ==> (false | (x1 =/= x2 & x1 @ y1 in A))                [prop-taut]
                   ==> (x1 =/= x2 & x1 @ y1 in A)                          [prop-taut]
                   ==> (x1 @ y1 in A)                                      [right-and]]))



define ms-lemma-1a  :=
pick-any x key val rest v
  assume hyp := (x =/= key)
     (!chain [([key _] ++ rest applied-to x = SOME v)
         <==> (rest applied-to x = SOME v) [apply-axioms]])

#define ms-lemma-1b :=
#  (forall m k v x . k @ v in map->set m & x =/= k ==> k @ v in map->set (m - x))

declare dom: (S, T) [(Map S T)] -> (Set.Set S)   [[alist->fmap]]

assert* dom-axioms :=
  [(dom empty-map = null)
   (dom [k _] ++ rest = k ++ dom rest)]

transform-output eval [Set.set->lst fmap->alist]

(eval dom ide-map)

conclude dom-lemma-1 :=
  (forall k v rest . k in dom [k v] ++ rest)
pick-any k v rest
  (!chain-> [true ==> (k in k ++ dom rest)      [Set.in-lemma-1]
                  ==> (k in dom [k v] ++ rest) [dom-axioms]])

conclude dom-lemma-2 :=
  (forall m k v . dom m subset dom [k v] ++ m)
pick-any m k v
  (!Set.subset-intro
     pick-any x
        (!chain [(x in dom m)
             ==> (x in k ++ dom m)       [Set.in-lemma-3]
             ==> (x in dom [k v] ++ m)   [dom-axioms]]))


conclude dom-characterization :=
  (forall m k . k in dom m <==> m applied-to k =/= NONE)
by-induction dom-characterization {
  (m as empty-map) =>
   pick-any k
     (!equiv
       (!chain [(k in dom m)
            ==> (k in null)                          [dom-axioms]
            ==> false                                [Set.NC]
            ==> (m applied-to k =/= NONE)            [prop-taut]])
        assume hyp := (m applied-to k =/= NONE)
          (!chain-> [true
```

```
               ==> (m applied-to k = NONE)         [apply-axioms]
               ==> (m applied-to k = NONE & hyp)   [augment]
               ==> false                           [prop-taut]
               ==> (k in dom m)                    [prop-taut]]))
| (m as (update (pair x y) rest)) =>
   let {IH := (forall k . k in dom rest <==> rest applied-to k =/= NONE)}
    pick-any k
      (!chain [(k in dom m)
          <==> (k in x ++ dom rest)                [dom-axioms]
          <==> (k = x | k in dom rest)             [Set.in-def]
          <==> (k = x | rest applied-to k =/= NONE) [IH]
          <==> (x = k | rest applied-to k =/= NONE) [sym]
          <==> (m applied-to k =/= NONE)           [apply-lemma-2]])
}

conclude dom-lemma-3 := (forall m k . dom (m - k) subset dom m)
by-induction dom-lemma-3 {
  (m as empty-map:(Map 'K 'V)) =>
     pick-any k:'K
       (!Set.subset-intro
          pick-any x:'K
           (!chain [(x in dom m - k)
               ==> (x in dom empty-map)   [remove-def]
               ==> (x in null)            [dom-axioms]
               ==> false                  [Set.NC]
               ==> (x in dom m)           [prop-taut]]))
| (m as (update (pair key:'K val:'V) rest)) =>
     pick-any k:'K
      let {IH := (!claim (forall k . dom rest - k subset dom rest));
           IH1 := (!chain-> [true ==> (dom rest - key subset dom rest) [IH]]);
           IH2 := (!chain-> [true ==> (dom rest - k subset dom rest) [IH]])}
       (!Set.subset-intro
          pick-any x:'K
            (!two-cases
               assume (key = k)
                 (!chain [(x in dom m - k)
                      ==> (x in dom m - key)       [(key = k)]
                      ==> (x in dom rest - key)    [remove-def]
                      ==> (x in dom rest)          [IH1 Set.SC]
                      ==> (x in key ++ dom rest)   [Set.in-lemma-3]
                      ==> (x in dom m)             [dom-axioms]])
               assume case-2 := (key =/= k)
                 (!chain [(x in dom m - k)
                      ==> (x in dom [key val] ++ (rest - k))  [remove-def]
                      ==> (x in key ++ dom rest - k)          [dom-axioms]
                      ==> (x = key | x in dom rest - k)       [Set.in-def]
                      ==> (x = key | x in dom rest)           [Set.SC IH2]
                      ==> (x in key ++ dom rest)              [Set.in-def]
                      ==> (x in dom m)                        [dom-axioms]]))))
}


declare size: (S, T) [(Map S T)] -> N [[alist->fmap]]

assert* size-axioms := [(size m = card dom m)]

transform-output eval [nat->int]

(eval size ide-map)

conclude ms-rec-lemma :=
  (forall m k v . size (m - k) < size [k v] ++ m)

conclude ms-rec-lemma
  pick-any m:(Map 'K 'V) key:'K val:'V
    let {L1 := (!by-contradiction (~ key in dom m - key)
                 assume h := (key in dom m - key)
                  (!absurd (!chain-> [true ==> ((m - key) applied-to key = NONE) [remove-correctness]])
                           (!chain-> [h ==> ((m - key) applied-to key =/= NONE)     [dom-characterization]])));
         L2 := (!chain-> [true ==> (key in dom [key val] ++ m) [dom-lemma-1]]);
```

```
        L3 := (!both (!chain-> [true ==> (dom m - key subset dom m) [dom-lemma-3]])
                     (!chain-> [true ==> (dom m subset dom [key val] ++ m) [dom-lemma-2]]));
        L4 := (!chain-> [L3 ==> (dom m - key subset dom [key val] ++ m) [Set.subset-transitivity]])}
     (!chain-> [L4 ==> (L4 & L2 & L1) [augment]
                   ==> (dom m - key proper-subset dom [key val] ++ m) [Set.proper-subset-lemma]
                   ==> (card dom m - key < card dom [key val] ++ m)   [Set.proper-subset-card-theorem]
                   ==> (size m - key < size [key val] ++ m)           [size-axioms]])

define ms-theorem :=
  (forall m k v . k @ v in map->set m <==> m applied-to k = SOME v)

(define (property m)
  (forall k v . k @ v in map->set m <==> m applied-to k = SOME v))

conclude ms-theorem
  (!strong-induction.measure-induction ms-theorem size
    pick-any m:(Map 'K 'V)
      assume IH := (forall m' . size m' < size m ==> property m')
        conclude (property m)
           datatype-cases (property m) on m  {
             (em as empty-map:(Map 'K 'V)) =>
               (pick-any k:'K v:'V
                 let {none := NONE:(Option 'V)}
                  (!equiv (!chain [(k @ v in map->set em)
                               ==> (k @ v in null)
                               ==> false
                               ==> (em applied-to k = SOME v)])
                          assume hyp := (em applied-to k = SOME v)
                            (!chain-> [true
                                   ==> (em applied-to k = none)                       [apply-axioms]
                                   ==> (em applied-to k = none & hyp)                 [augment]
                                   ==> (em applied-to k = none & em applied-to k =/= none) [option-results]
                                   ==> false                                  [prop-taut]
                                   ==> (k @ v in map->set em)                 [prop-taut]])))
           | (map as (update (pair key:'K val:'V) rest)) =>
              pick-any k:'K v:'V
               let {goal := (k @ v in map->set map <==> map applied-to k = SOME v);
                    lemma := (!chain-> [true ==> (size rest - key < size map) [ms-rec-lemma]
                                             ==> (size rest - key < size m)   [(m = map)]])}
                 (!two-cases
                   assume case1 := (k = key)
                     (!equiv assume hyp := (k @ v in map->set map)
                               let {D := (!chain-> [hyp
                                                ==> (k @ v in key @ val ++ map->set rest - key)
[map->set-def]
                                                ==> (key @ v in key @ val ++ map->set rest - key) [case1]
                                                ==> (key @ v = key @ val | key @ v in map->set rest - key) [Set.in-de
                                  (!cases D
                                     assume h1 := (key @ v in map->set rest - key)
                                       let {_ := (!absurd (!chain-> [h1 ==> ((rest - key) applied-to key = SOME v)
[IH]
                                                                        ==> (NONE = SOME v)
[remove-correctness]])
                                                          (!chain-> [true ==> (NONE =/= SOME v) [option-results]]))}
                                         (!from-false (map applied-to k = SOME v))
                                     assume h2 := (key @ v = key @ val)
                                       let {v=val := (!chain [h2 ==> (v = val)])}
                                        (!chain-> [(map applied-to key) = (SOME val) [apply-axioms]
                                                                        = (SOME v)    [v=val]
                                               ==> (map applied-to key = SOME v)
                                               ==> (map applied-to k = SOME v)   [(k = key)]]))
                            assume hyp := (map applied-to k = SOME v)
                              let {val=v := (!chain-> [(SOME val)
                                                   = (map applied-to key) [apply-axioms]
                                                   = (map applied-to k)    [(k = key)]
                                                   = (SOME v)     [hyp]
                                               ==> (val = v)      [option-results]])}
                                  (!chain-> [true ==> (key @ val in key @ val ++ map->set (rest - key)) [Set.in-lemma-
                                                  ==> (key @ val in map->set map)
[map->set-def]
```

```
                                                  ==> (k @ val  in map->set map)
[(k = key)]
                                                  ==> (k @ v in map->set map)
[val=v]]))
                    assume case2 := (k =/= key)
                      (!iff-comm
                        (!chain [(map applied-to k = SOME v)
                            <==> (rest applied-to k = SOME v)                        [apply-axioms]
                            <==> ((rest - key) applied-to k = SOME v)                [remove-correctness-2]
                            <==> (k @ v in map->set rest - key)              [IH]
                            <==> (k @ v in key @ val ++ map->set rest - key) [(k @ v in map->set rest - key <==>
                                                                               k @ v in key @ val ++ map->set rest - k
                                                                              <== case2 [opair-lemma]]
                            <==> (k @ v in map->set map)                     [map->set-def]])))
           })

(eval dom ide-map)

conclude dom-characterization-2 :=
  (forall m x . x in dom m <==> exists v . x @ v in map->set m)
pick-any m:(Map 'K 'V) x:'K
   (!chain [(x in dom m)
       <==> (m applied-to x =/= NONE)                [dom-characterization]
       <==> (exists v . m applied-to x = SOME v)     [option-results]
       <==> (exists v . x @ v in map->set m) [ms-theorem]])

conclude ms-corollary :=
  (forall m k . m applied-to k = NONE <==> ~ exists v . k @ v in map->set m)
pick-any m:(Map 'K 'V) k:'K
  (!equiv (!chain [(m applied-to k = NONE)
               ==> (~ exists v . m applied-to k = SOME v)     [option-results]
               ==> (~ exists v . k @ v in map->set m) [ms-theorem]])
          (!chain [(~ exists v . k @ v in map->set m)
               ==> (~ exists v . m applied-to k = SOME v)     [ms-theorem]
               ==> (m applied-to k = NONE)                    [option-results]]))


conclude identity-characterization-1 :=
  (forall m1 m2 . m1 = m2 ==> forall k . m1 applied-to k = m2 applied-to k)
pick-any m1:(Map 'S 'T) m2:(Map 'S 'T)
  assume hyp := (m1 = m2)
   let {m1=m2 := (!chain-> [hyp ==> (map->set m1 = map->set m2) [map-identity]])}
    pick-any k:'S
      (!cases (!chain-> [true ==> (m1 applied-to k = NONE | exists v . m1 applied-to k = SOME v) [option-results]])
        assume case1 := (m1 applied-to k = NONE)
          let {p := (!by-contradiction (m2 applied-to k = NONE)
                      assume h := (m2 applied-to k =/= NONE)
                        pick-witness v for (!chain-> [h ==> (exists v . m2 applied-to k = SOME v) [option-results]]) w
                          (!chain-> [wp ==> (k @ v in map->set m2)     [ms-theorem]
                                        ==> (k @ v in map->set m1)     [m1=m2]
                                        ==> (m1 applied-to k = SOME v)         [ms-theorem]
                                        ==> (m1 applied-to k =/= NONE)         [option-results]
                                        ==> (case1 & m1 applied-to k =/= NONE) [augment]
                                        ==> false                      [prop-taut]]))}
            (!combine-equations (m1 applied-to k = NONE) (m2 applied-to k = NONE))
        assume case2 := (exists v . m1 applied-to k = SOME v)
          pick-witness v for case2
            (!combine-equations
               (m1 applied-to k = SOME v)
               (!chain-> [(m1 applied-to k = SOME v)
                      ==> (k @ v in map->set m1)  [ms-theorem]
                      ==> (k @ v in map->set m2)  [m1=m2]
                      ==> (m2 applied-to k = SOME v)      [ms-theorem]])))

conclude identity-characterization-2 :=
  (forall m1 m2 . (forall k . m1 applied-to k = m2 applied-to k) ==> m1 = m2)
pick-any m1:(Map 'S 'T) m2:(Map 'S 'T)
  assume hyp := (forall k . m1 applied-to k = m2 applied-to k)
   let {m1=m2-as-sets :=
        (!Set.set-identity-intro-direct
```

```
          (!pair-converter
             pick-any k:'S v:'T
               (!chain [(k @ v in map->set m1)
                   <==> (m1 applied-to k = SOME v)       [ms-theorem]
                   <==> (m2 applied-to k = SOME v)       [hyp]
                   <==> (k @ v in map->set m2)   [ms-theorem]]))))}
      (!chain-> [m1=m2-as-sets ==> (m1 = m2) [map-identity]])

conclude identity-characterization :=
  (forall m1 m2 . m1 = m2 <==> forall k . m1 applied-to k = m2 applied-to k)
pick-any m1:(Map 'S 'T) m2:(Map 'S 'T)
  (!equiv
     (!chain [(m1 = m2) ==> (forall k . m1 applied-to k = m2 applied-to k) [identity-characterization-1]])
     (!chain [(forall k . m1 applied-to k = m2 applied-to k) ==> (m1 = m2) [identity-characterization-2]]))


declare restricted-to: (S, T) [(Map S T) (Set.Set S)] -> (Map S T) [150 |^ [alist->fmap Set.lst->set]]

assert* restrict-axioms :=
   [(empty-map |^ _ = empty-map)
    (k in A ==> [k v] ++ rest |^ A = [k v] ++ (rest |^ A))
    (~ k in A ==> [k v] ++ rest |^ A = rest |^ A)]

(eval [[1 --> 'a] [2 --> 'b] [3 --> 'c]] |^ [1 3])

conclude restriction-theorem-1 := (forall m A . dom m |^ A subset A)
by-induction restriction-theorem-1 {
  empty-map =>
    pick-any A
      (!Set.subset-intro
        pick-any x
          (!chain [(x in dom empty-map |^ A)
               ==> (x in dom empty-map)      [restrict-axioms]
               ==> (x in null)              [dom-axioms]
               ==> false                     [Set.NC]
               ==> (x in A)                  [prop-taut]]))
| (m as (update (pair k v) rest)) =>
     pick-any A
     let {IH := (forall A . dom rest |^ A subset A);
          lemma := (!chain-> [true ==> (dom rest |^ A subset A) [IH]])}
       (!two-cases
          assume case-1 := (k in A)
            (!Set.subset-intro
               pick-any x
                 (!chain [(x in dom m |^ A)
                      ==> (x in dom [k v] ++ (rest |^ A)) [restrict-axioms]
                      ==> (x in k ++ dom rest |^ A)        [dom-axioms]
                      ==> (x = k | x in dom rest |^ A)     [Set.in-def]
                      ==> (x in A | x in dom rest |^ A)    [case-1]
                      ==> (x in A | x in A)               [Set.SC]
                      ==> (x in A)                        [prop-taut]]))
          assume case-2 := (~ k in A)
            (!Set.subset-intro
               pick-any x
                 (!chain [(x in dom m |^ A)
                      ==> (x in dom rest |^ A) [restrict-axioms]
                      ==> (x in A)            [Set.SC]]))))
}


conclude restriction-theorem-2 :=
  (forall m A . dom m subset A ==> m |^ A = m)
by-induction restriction-theorem-2 {
  (m as empty-map) =>
    pick-any A
      assume hyp := (dom m subset A)
        (!chain [(m |^ A) = m [restrict-axioms]])
| (m as (update (pair key val) rest)) =>
    pick-any A
      assume hyp := (dom m subset A)
```

```
        let {lemma1 := (!chain-> [true ==> (key in dom m) [dom-lemma-1]
                                   ==> (key in A)     [Set.SC]]);
             lemma2 := (!chain-> [true ==> (dom rest subset dom m)       [dom-lemma-2]
                                   ==> (dom rest subset dom m & hyp) [augment]
                                   ==> (dom rest subset A)           [Set.subset-transitivity]]);
             IH := (forall A . dom rest subset A ==> rest |^ A = rest)}
          (!chain [(m |^ A)
                 = ([key val] ++ (rest |^ A))   [restrict-axioms]
                 = ([key val] ++ rest)          [IH]])
}


declare range: (S, T) [(Map S T)] -> (Set.Set T)   [[alist->fmap]]

assert* range-def :=
  [(range m = Set.range map->set m)]

(eval range ide-map)

conclude range-lemma-1 :=
  (forall m v . v in range m <==> exists k . k @ v in map->set m)
pick-any m v
  (!chain [(v in range m)
      <==> (v in Set.range map->set m)     [range-def]
      <==> (exists k . k @ v in map->set m) [Set.range-characterization]])

conclude range-characterization :=
  (forall m v . v in range m <==> exists k . m at k = SOME v)
pick-any m v
  (!chain [(v in range m)
      <==> (exists k . k @ v in map->set m) [range-lemma-1]
      <==> (exists k . m at k = SOME v)     [ms-theorem]])

conclude range-lemma-2 :=
   (forall k v rest . v in range [k v] ++ rest)
pick-any k v rest
  (!chain<- [(v in range [k v] ++ rest)
          <== (v in Set.range map->set [k v] ++ rest)     [range-def]
          <== (v in Set.range k @ v ++ map->set rest - k) [map->set-def]
          <== (v in v ++ Set.range map->set rest - k)     [Set.range-def]
          <== (v = v | v in Set.range map->set rest - k)  [Set.in-def]
          <== (v = v)                                      [alternate]])

define range-lemma-conjecture :=
    (forall m k v . range m subset range [k v] ++ m)

(falsify range-lemma-conjecture 10)

conclude removal-range-theorem :=
  (forall m k . range m - k subset range m)
pick-any m k
  (!Set.subset-intro
     pick-any v
       assume hyp := (in v range m - k)
         pick-witness key for
           (!chain<- [(exists key . m - k at key = SOME v)
                  <== hyp [range-characterization]])
           key-premise
           let {k!=key :=
                  (!by-contradiction (k =/= key)
                    assume (k = key)
                      (!absurd (!chain-> [key-premise
                                  ==> (m - key at key = SOME v) [(k = key)]])
                               (!chain-> [true ==> (m - key at key = NONE) [remove-correctness]
                                               ==> (m - key at key =/= SOME v) [option-results]])))}
             (!chain-> [k!=key ==> (m - k at key = m at key) [remove-correctness-2]
                           ==> (SOME v = m at key)       [key-premise]
                           ==> (m at key = SOME v)       [sym]
                           ==> (exists key . m at key = SOME v) [existence]
                           ==> (v in range m)                   [range-characterization]]))
```

```
declare range-restricted: (S, T) [(Map S T) (Set.Set T)] -> (Map S T) [150 ^| [alist->fmap Set.lst->set]]

assert* range-restricted-def :=
  [(empty-map ^| _ = empty-map)
   ([k v] ++ rest ^| A = [k v] ++ (rest - k ^| A) <== v in A)
   ([k v] ++ rest ^| A = rest - k ^| A <== ~ v in A)]

(define p   (forall m A . range m ^| A subset range m))
define eye-color := [['bob --> 'brown]  ['tom --> 'blue] ['lisa --> 'green]
                     ['peter --> 'blue] ['ann --> 'brown]]

(eval eye-color ^| ['blue])

(define vpf
  (method (goal premises)
    (!vprove-from goal premises [['poly true] ['subsorting false] ['max-time 3000]])))

(define spf
  (method (goal premises)
    (!sprove-from goal premises [['poly true] ['subsorting false] ['max-time 300]])))

### CAUTION: THE PATTERN (m as null) seemed to work!

define range-restriction-theorem-1 :=
  (forall m A . range m ^| A subset range m)

declare agree-on: (S, T) [(Map S T) (Map S T) (Set.Set S)] -> Boolean
                         [[alist->fmap alist->fmap Set.lst->set]]

assert* agree-on-def := [((agree-on m1 m2 A) <==> m1 |^ A = m2 |^ A)]

(eval (agree-on ide-map ide-map ['a 'b]))

(eval (agree-on [['a --> 1] ['b --> 2]]
                [['b --> 3] ['a --> 1]]
                ['b]))

declare override: (S, T) [(Map S T) (Map S T)] -> (Map S T) [** [alist->fmap alist->fmap]]

assert* override-def :=
  [(m ** [] = m)
   (m ** [k v] ++ rest = [k v] ++ (m ** rest))]

(eval [[1 --> 'a] [2 --> 'b]] ** [[1 --> 'foo] [3 --> 'c]])


conclude override-theorem-1 := (forall m . [] ** m = m)
by-induction override-theorem-1 {
  (m as empty-map) =>
    (!chain [(empty-map ** m) = empty-map [override-def]])
| (m as (update (pair k v) rest)) =>
   let {IH := ([] ** rest = rest)}
    (!chain [(empty-map ** m)
           = ([k v] ++ (empty-map ** rest))  [override-def]
           = ([k v] ++ rest)                 [IH]])
}

define conj1 := (forall m1 m2 . dom m2 ** m1 = (dom m2) \/ (dom m1))

by-induction (forall m1 m2 . dom m2 ** m1 = (dom m2) \/ (dom m1)) {
  (m1 as empty-map:(Map 'K 'V)) =>
    pick-any m2:(Map 'K 'V)
      (!chain [(dom m2 ** m1)
             = (dom m2)                                [override-def]
             = (null \/ dom m2)                        [Set.union-def]
             = ((dom m2) \/ null)                      [Set.union-commutes]
             = ((dom m2) \/ (dom m1))                  [dom-axioms]])
 | (m1 as (update (pair k:'K v:'V) rest)) =>
    let {IH := (forall m2 . dom m2 ** rest = (dom m2) \/ (dom rest))}
```

```
    pick-any m2:(Map 'K 'V)
      (!chain [(dom m2 ** m1)
             = (dom [k v] ++ (m2 ** rest))      [override-def]
             = (k ++ dom (m2 ** rest))          [dom-axioms]
             = (k ++ ((dom m2) \/ (dom rest))) [IH]
             = ((dom m2) \/ k ++ dom rest)      [Set.union-lemma-2]
             = ((dom m2) \/ dom m1)             [dom-axioms]])
}

define conj2 :=
  (forall m1 m2 k . k in dom m1 ==> (m2 ** m1) applied-to k = m1 applied-to k)


# (falsify conj2 20)

by-induction conj2 {
  (m1 as empty-map:(Map 'S 'T)) =>
    pick-any m2:(Map 'S 'T) k:'S
      (!chain [(k in dom m1)
            ==> (k in null)    [dom-axioms]
            ==> false          [Set.NC]
            ==> ((m2 ** m1) applied-to k = m1 applied-to k) [prop-taut]])
| (m1 as (update (pair key val) rest)) =>
    let {IH := (forall m2 k . k in dom rest ==> (m2 ** rest) applied-to k = rest applied-to k)}
      pick-any m2 k
        assume hyp := (k in dom m1)
          (!cases (!chain-> [hyp
                        ==> (k in key ++ dom rest)     [dom-axioms]
                        ==> (k = key | k in dom rest)  [Set.in-def]
                        ==> (k = key | k =/= key & k in dom rest)  [prop-taut]])
             assume (k = key)
               (!chain [((m2 ** m1) applied-to k)
                      = ([key val] ++ (m2 ** rest) applied-to k)   [override-def]
                      = ([key val] ++ (m2 ** rest) applied-to key) [(k = key)]
                      = (SOME val)                                 [apply-axioms]
                      = (m1 applied-to key)                        [apply-axioms]
                      = (m1 applied-to k)                          [(k = key)]])
             assume (k =/= key & k in dom rest)
                 (!chain [((m2 ** m1) applied-to k)
                        = (([key val] ++ (m2 ** rest)) applied-to k)   [override-def]
                        = ((m2 ** rest) applied-to k)                  [apply-axioms]
                        = (rest applied-to k)                          [IH]
                        = (m1 applied-to k)                            [apply-axioms]]))
}

define conj3 := (forall m1 m2 . range m2 ** m1 = (range m2) \/ (range m1))
(falsify conj3 10)
(falsify conj3 20)

conclude restrict-theorem-3 :=
  (forall m2 m1 A . (m1 ** m2) |^ A = m1 |^ A ** m2 |^ A)
by-induction restrict-theorem-3 {
  (m2 as empty-map) =>
    pick-any m1 A
      (!combine-equations
          (!chain [((m1 ** m2) |^ A) =  (m1 |^ A)])
          (!chain [(m1 |^ A ** m2 |^ A)
                 = (m1 |^ A ** empty-map)
                 = (m1 |^ A)]))
| (m2 as (update (pair k v) rest)) =>
   let {IH := (forall m1 A . (m1 ** rest) |^ A = m1 |^ A ** rest |^ A)}
    pick-any m1 A
      (!two-cases
         assume (k in A)
           (!combine-equations
              (!chain [((m1 ** m2) |^ A)
                     = (([k v] ++ (m1 ** rest)) |^ A)     [override-def]
                     = ([k v] ++ ((m1 ** rest) |^ A))     [restrict-axioms]
                     = ([k v] ++ (m1 |^ A ** rest |^ A)) [IH]])
              (!chain [(m1 |^ A ** m2 |^ A)
```

```
                         = (m1 |^ A ** [k v] ++ (rest |^ A)) [restrict-axioms]
                         = ([k v] ++ (m1 |^ A ** rest |^ A)) [override-def]]))
          assume (~ k in A)
            (!chain [((m1 ** m2) |^ A)
                     = (([k v] ++ (m1 ** rest)) |^ A)       [override-def]
                     = ((m1 ** rest) |^ A)                  [restrict-axioms]
                     = (m1 |^ A ** rest |^ A)               [IH]
                     = (m1 |^ A ** m2 |^ A)                 [restrict-axioms]]))
}

declare compose: (S1, S2, S3) [(Map S2 S3) (Map S1 S2)] -> (Map S1 S3) [o [alist->fmap alist->fmap]]

assert* compose-def :=
  [(_ o empty-map = empty-map)
   (m o [k v] ++ more = [k v'] ++ (m o more) <== m applied-to v = SOME v')
   (m o [k v] ++ more = m o more <== m applied-to v = NONE)]

(define M1 [[1 --> 'a] [2 --> 'b] [1 --> 'c]])
(define M2 [['a --> true] ['b --> false] ['foo --> true]])
(eval M2 o M1)

define capitals :=
  [['paris --> 'france] ['tokyo --> 'japan] ['cairo --> 'egypt]]

define countries :=
  [['france --> 'europe] ['algeria --> 'africa] ['japan --> 'asia]]

(eval countries o capitals)

(define [t1 t2] [(alist->fmap M2) (alist->fmap M1)])
#(c t1 t2)

define composition-is-comm := (forall m1 m2 . m1 o m2 = m2 o m1)

(falsify composition-is-comm 20)

define composition-is-assoc := (forall m1 m2 m3 . m1 o (m2 o m3) = (m1 o m2) o m3)
#(falsify composition-is-assoc 20)

#(falsify (close (((m3 o m2) o m1) = m3 o (m2 o m1))) 20)

define [n] := [?n:N]

declare iterate: (S, S) [(Map S S) N] -> (Map S S) [^^ [alist->fmap int->nat]]
define [^^ iterated] := [iterate iterate]

assert* iterate-axioms :=
        [(m ^^ zero = m)
         (m ^^ succ n =  m o (m ^^ n))]

let {m := (alist->fmap [[1 --> 2] [2 --> 3] [3 --> 1]]);
     _ := (print "\nm iterated once: " (eval map->set m ^^ 1));
     _ := (print "\nm iterated twice: " (eval map->set m ^^ 2));
     _ := (print "\nm iterated thrice: " (eval map->set m ^^ 3))}
  (print "\nAre m and m^^3 identical?: " (eval m = m ^^ 3))

declare compose2: (S) [(Map S S) (Map S S)] -> (Map S S) [[alist->fmap alist->fmap]]

assert* compose2-def :=
  [(m compose2 empty-map = m)
   (m compose2 [k v] ++ more = [k v'] ++ (m compose2 more) <== m applied-to v = SOME v')
   (m compose2 [k v] ++ more = [k v] ++ (m compose2 more) <== m applied-to v = NONE)]

define comp2-is-comm := (forall m1 m2 . m1 compose2 m2 = m2 compose2 m1)

(falsify comp2-is-comm 10)

define comp2-is-assoc := (forall m1 m2 m3 . m1 compose2 (m2 compose2 m3) = (m1 compose2 m2) compose2 m3)

#(falsify comp2-is-assoc 80)
```

```
(define comp2-app-lemma
   (forall m1 m2 k v . (m2 compose2 m1) applied-to k = SOME v <==>
                        ((exists v' . m1 applied-to k = SOME v' & m2 applied-to v' = SOME v) |
                         (m1 applied-to k = NONE & m2 applied-to k = SOME v))))

# (falsify comp2-app-lemma 10)

declare compatible: (S, T) [(Map S T) (Map S T)] -> Boolean [<-> [alist->fmap alist->fmap]]

assert* compatible-def :=
  [(m1 <-> m2 <==> agree-on m1 m2 (dom m1) /\ (dom m2))]

pick-any m
  (!chain<- [(m <-> m)
        <== (agree-on m m (dom m) /\ (dom m))  [compatible-def]
        <== (agree-on m m dom m)               [Set.intersection-lemma-3]
        <== (m |^ dom m = m |^ dom m)          [agree-on-def]])

(eval [[1 --> 'a] [2 --> 'b]] <-> [[1 --> 'a] [3 --> 'c]])

(eval [[1 --> 'a] [2 --> 'b]] <-> [[1 --> 'a] [2 --> 'foo] [3 --> 'c]])

define compatible-theorem-1 := (forall m . m <-> m)

(falsify compatible-theorem-1 20)

define compatible-theorem-2 := (forall m1 m2 . m1 <-> m2 <==> m2 <-> m1)

#(running-time (lambda () (falsify compatible-theorem-2 10)) 0)
# with new eval1: 4.22

define compatible-theorem-3 := (forall m1 m2 m3 . m1 <-> m2 & m2 <-> m3 ==> m1 <-> m3)

(falsify compatible-theorem-3 10)

}


EOF
load "c:\\np\\lib\\basic\\fmaps"
```