```
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.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] :=
define [m m1 m2 m3 rest rest1] := [?m:(Map 'S1 'S2) ?m1:(Map 'S1 'S2) ?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]
define [M := [[1 --> 'a] [2 --> 'b] [1 --> 'c]]]

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

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

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

#assert* remove-axioms :=
  [(_ removed-from empty-map = empty-map)
   (key removed-from [key _] ++ rest = key removed-from rest)
   (key /= x ==> x removed-from 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 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 t2 (- ?x null))
define t3 (- ?x ?y))

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

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

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

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

#assert* remove-axioms :=
  [(_ removed-from empty-map = empty-map)
   (key removed-from [key _] ++ rest = key removed-from rest)
   (key /= x ==> x removed-from 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 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 t2 (- ?x null))
define t3 (- ?x ?y))

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

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

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

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

define (key removed-from map) := (key removed-from map)
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) [option-results]
  (!chain [key /= x])
    ==> (m at x = rest at x) [option-results])

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
  (!equiv
     case-1 := (k = x)
     (!equiv
        hyp := ([k v] ++ rest applied-to x =/= NONE)
        (!chain-> [(k = x) ==> (k = x | rest applied-to x =/= NONE)]))
     (!equiv
        case-2 := (k /= x)
        (!equiv
           hyp := ([k v] ++ rest applied-to x =/= NONE)
           (!chain-> [(k =/= x)])

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
  (!equiv
     hyp := (m applied-to k = SOME v1 & m applied-to k = SOME v2)
     ![chain-> ![SOME v1]]
     ![chain-> ![SOME v2]]
     ![chain-> ![v1 = v2]]

conclude remove-correctness :=
(by-induction remove-correctness {
  (m as empty-map) =>
  pick-any x
  ![chain ![x applied-to x]]
  ![chain ![x applied-to x]]
  ![chain ![x applied-to x]]
| (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]]
  ![chain ![m - x applied-to x]]
  ![chain ![m - x applied-to x]]
```
lib/basic/fmaps.ath

208  = (m - key applied-to key) [case1]
209  = (rest - x applied-to x) [case1 remove-def]
210  = NONE          [IH])

211  assume case2 := (key /= x)
212  (!chain [ (m - x applied-to x)
213    = ([key val] ++ (rest - x) applied-to x) [remove-def]
214    = (rest - x applied-to x)                [apply-axioms]
215    = NONE                 [IH])])

216  }

217  define (RC2-M goal p1 p2) :=
218    match [goal p1 p2] {
219      [(~ (s = t)) (s = u) (~ (u = t))] =>
220        (!by-contradiction goal
221         assume (~ goal)
222         (!chain-~ [ (~ goal)]
223           => (s = t)  [dn]
224           => (u = t) [ (s = u)]
225           => (u = t & u /= t) [augment]
226           => false [prop-taut]])
227      | (m as empty-map) =>
228        pick-any x y
229          assume hyp := (x =/= y)
230          (!chain-~ [ (m - x) at y]
231            = (m at y) [remove-def])
232      | (m as (update (pair key val) rest)) =>
233        let {IH := (forall x y . x /= y => (rest - x) at y = rest at y)}
234        pick-any x y
235          assume hyp := (x =/= y)
236          (!two-cases
237            assume (key = x)
238              #let {lemma := (!CongruenceClosure.cc (key =/= y) [case1 hyp])}
239              let {lemma := ((RC2-M (key =/= y) casel hyp))
240                (!chain [ (((m - x) at y)
241                  = ((rest - x) at y) [key = x] remove-def]
242                  = (rest at y)            [IH]
243                  = (m at y)              [apply-axioms]])
244            assume (key =/= x)
245              (!two-cases
246                assume (key = y)
247                (!combine-equations
248                  (!chain [ (((m - x) at y)
249                    = (((key val) ++ (rest - x)) at y) [remove-def]
250                    = (SOME val)                [apply-axioms]])
251                !chain [ (m at y)
252                  = (SOME val)                [apply-axioms]])
253            assume (key =/= y)
254                (!combine-equations
255                  (!chain [ (((m - x) at y)
256                    = (((key val) ++ (rest - x)) at y) [remove-def]
257                    = (rest - x) at y)     [apply-axioms]
258                    = (rest at y)            [IH])])
259                !chain [ (m at y)
260                  = (rest at y)            [apply-axioms]])})
261            })
262          }
263        }
264        conclude remove-correctness-2 :=
265          (forall m x y . x /= y => (m - x) at y = m at y)
266        by-induction remove-correctness-2 {
267          (m as empty-map) =>
268            pick-any x y
269              assume hyp := (x =/= y)
270              (!chain-~ [ ((m - x) at y]
271                = (m at y) [remove-def])
272          | (m as (update (pair key val) rest)) =>
273            let {IH := (forall x y . x /= y => (rest - x) at y = rest at y)}
274            pick-any x y
275              assume hyp := (x =/= y)
276              (!two-cases
277                assume (key = x)
278                #let {lemma := (!CongruenceClosure.cc (key =/= y) [case1 hyp])}
279                let {lemma := ((RC2-M (key =/= y) casel hyp))
280                  (!chain [ (((m - x) at y)
281                    = ((rest - x) at y) [key = x] remove-def]
282                    = (rest at y)            [IH]
283                    = (m at y)              [apply-axioms]])
284                assume (key =/= x)
285                (!two-cases
286                  assume (key = y)
287                  (!combine-equations
288                    (!chain [ (((m - x) at y)
289                      = (((key val) ++ (rest - x)) at y) [remove-def]
290                      = (SOME val)                [apply-axioms]])
291                    !chain [ (m at y)
292                      = (SOME val)                [apply-axioms]])
293                  assume (key =/= y)
294                  (!combine-equations
295                    (!chain [ (((m - x) at y)
296                      = (((key val) ++ (rest - x)) at y) [remove-def]
297                      = (rest - x) at y)     [apply-axioms]
298                      = (rest at y)            [IH])])
299                    !chain [ (m at y)
300                      = (rest at y)            [apply-axioms]])})
301                })
302          }
303          declare map->set: (S, T) [(Map S T)] -> (Set.Set (Pair S T)) [[alist->fmap]]
304          assert* map->set-def :=
305            [ (map->set empty-map = null)
306             (map->set [k v] ++ rest = (k & v) ++ map->set rest - k)]
307          assert* map-identity := (ml = m2 <==> map->set ml = map->set m2)
308```
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 (eval map->set ide-map)
  (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
  (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
)

conclude dom-lemma-1 :=
(forall m k v x . k v in map->set m & x /= k ==> k v in map->set (m - x))

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))

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

transform-output eval [Set.set->lst fmap->alist]
conclude dom-lemma :=
(forall k v rest . k in dom [k v] ++ rest)

conclude dom-lemma-2 :=
(forall m k v . dom m subset dom [k v] ++ m)

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 (eval (alist->fmap ide-map)
    (eval map->set ide-map)
    (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
    (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
  )
  [k in dom m]
  [k in dom [k v] ++ m]
  (equiv (eval (alist->fmap ide-map)
    (eval map->set ide-map)
    (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
    (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
  )
  [k in null]
  false
  [m applied-to k =/= NONE]
  [prop-taut])
  assume hyp := (m applied-to k =/= NONE)
  (equiv (eval (alist->fmap ide-map)
    (eval map->set ide-map)
    (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
    (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
  )
  [true]
  [true]
lib/basic/fmaps.ath

6

===> (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

| (k in dom m) |

<==> (k in x ++ dom rest) [dom-axioms]

<==> (x = k | rest applied-to k =/= NONE) [IH]

<==> (x = k | rest applied-to k =/= NONE) [sym]

<==> (m applied-to k =/= NONE) [apply-lemma-2]]

| (m as (pair x y) rest)) =>

let [IH := (forall k . k in dom rest <=> rest applied-to k =/= NONE)]

pick-any k

| (k in dom m) |

<==> (k in dom m - k) [dom-axioms]

<==> (x in dom rest - k) [Set.in-def]

<==> (x = k | rest applied-to k =/= NONE) [IH]

<==> (k in dom rest - k) [sym]

<==> (m applied-to k =/= NONE) [apply-lemma-2]]

let [IH := (forall k . k in dom rest <=> rest applied-to k =/= NONE)]

pick-any k

| (k in dom m) |

<==> (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 val) rest)) =>

pick-any k

| (k in dom m - k) |

<==> (x in dom m - k) [dom-axioms]

<==> (x in dom rest) [IH1 Set.SC]

<==> (x in dom m) [dom-axioms])

| (m as (pair key val) rest)) =>

pick-any k

| (k in dom m - k) |

<==> (x in dom [key val] ++ (rest - k)) [remove-def]

<==> (x in key ++ dom rest - k) [dom-axioms]

<==> (x in key ++ dom rest) [Set.in-lemma-3]

<==> (x in dom m) [dom-axioms])

assume case-2 := (key =/= k)

| (k in dom m - k) |

<==> (x in dom m - key) [(key = k)]

<==> (x in key ++ dom rest - k) [dom-axioms]

<==> (x in key ++ dom rest) [Set.in-def]

<==> (x in dom m) [dom-axioms])

assume case-2 := (key =/= k)

| (k in dom m - k) |

<==> (x in dom [key val] ++ (rest - k)) [remove-def]

<==> (x in key ++ dom rest - k) [dom-axioms]

<==> (x in key ++ dom rest) [Set.in-def]

<==> (x in dom m) [dom-axioms])

declare size: (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 [key val] ++ 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-> [true ==> (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-> [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))
(lequiv (!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 casel := (k = key)
(lequiv assume hyp := (k @ v in map->set map)
let (D := (!chain-> [hyp
   ==> (k @ v in key @ val ++ map->set rest - key)
   ==> (key @ v in key @ val ++ map->set rest - key) [casel]
   ==> (key @ v = key @ val | key @ v in map->set rest - key) [Set.in-def]
(!cases D
assume h1 := (key @ v in map->set rest - key)
let (_ := (!absurd (!chain-> [h1 ==> ((rest - key) applied-to key = SOME v)
   ==> (NONE =/= SOME v)
   [remove-correctness]))
   [| (map as update (pair key:'K val:'V) rest)) =>
})]}

\text{lib/basic/fmaps.ath}
lib/basic/fmaps.ath

484 => (k @ val in map->set map)
485 => (k @ v in map->set map)

486 [(k = key)]
487 => (k @ v in map->set map)
488 [val=v])

489 assume case2 := (k /= key)

490 (!iff-comm
491 (!chain [(map applied-to k = SOME v)
492 ==> (rest applied-to k = SOME v) [apply-axioms]
493 ==> (rest - key) applied-to k = SOME v [remove-correctness-2]
494 (k @ v in key @ val ++ map->set rest - key) [(k @ v in map->set rest - key <=>
495 k @ v in key @ val ++ map->set rest
496 <= case2 [opair-lemma]])
497 (k @ v in map->set map) [map->set-def]])

498}

499 |(eval dom ide-map)

|f|f|f|

500 conclude dom-characterization-2 :=
501 (forall m x . x in dom m <===> exists v . x @ v in map->set m)

502 pick-any m:(Map 'K 'V) x:'K

503 (!equiv (!chain [(x in dom m)
504 ==> (m applied-to x =/= NONE) [dom-characterization]
505 ==> (exists v . m applied-to x = SOME v) [option-results]
506 ==> (exists v . x @ v in map->set m) [ms-theorem]])

507 conclude ms-corollary :=
508 (forall k . m applied-to k = NONE <===> ~ exists v . k @ v in map->set m)

509 pick-any m:(Map 'K 'V) k:'K

510 (true ==> (m applied-to k = NONE)
511 ==> (~ exists v . m applied-to k = SOME v) [option-results]
512 ==> (~ exists v . k @ v in map->set m) [ms-theorem]])

513 !equiv (!chain [(~ exists v . k @ v in map->set m)
514 ==> (~ exists v . m applied-to k = SOME v) [ms-theorem]
515 ==> (m applied-to k = NONE) [option-results]])

516 |f|f|f|

517 conclude identity-characterization-1 :=
518 (forall m1 m2 . m1 = m2 ==> forall k . m1 applied-to k = m2 applied-to k)

519 pick-any m1:(Map 'S 'T) m2:(Map 'S 'T)

520 assume hyp := (m1 = m2)

521 let [ml=m2 := (!chain-> [hyp ==> (map->set ml = map->set m2) [map-identity]])]

522 pick-any k:'S

523 (!cases (!chain-> [true ==> (ml applied-to k = NONE | exists v . ml applied-to k = SOME v) [option-results]])
524 assume casel := (ml applied-to k = NONE)

525 let [p := (by-contradiction (m2 applied-to k =/= NONE)]

526 assume h := (m2 applied-to k =/= NONE)

527 pick-witness v for (!chain-> [h ==> (exists v . ml applied-to k = SOME v) [option-results]])

528 (!chain-> [wp ==> (k @ v in map->set m2) [ms-theorem]
529 ==> (k @ v in map->set ml) [ml=m2]
530 ==> (ml applied-to k = SOME v) [ms-theorem]
531 ==> (m applied-to k =/= NONE) [option-results]
532 ==> (casel & ml applied-to k =/= NONE) [augment]
533 ==> false [prop-taut]])])

534 (!combine-equations (ml applied-to k = NONE) (m2 applied-to k =/= NONE))

535 assume case2 := (exists v . ml applied-to k = SOME v)

536 pick-witness v for case2

537 (!combine-equations
538 (ml applied-to k = SOME v)
539 (!chain-> [(ml applied-to k = SOME v)
540 ==> (k @ v in map->set ml) [ms-theorem]
541 ==> (k @ v in map->set m2) [ml=m2]
542 ==> (m2 applied-to k = SOME v) [ms-theorem]])

543 conclude identity-characterization-2 :=
544 (forall m1 m2 . (forall k . ml applied-to k = m2 applied-to k) ==> ml = m2)

545 pick-any m1:(Map 'S 'T) m2:(Map 'S 'T)

546 assume hyp := (forall k . ml applied-to k = m2 applied-to k)

547 let [ml=m2-as-sets :=

548 (!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]]))

(conclude identity-characterization :=
  (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 ++ [v] ++ (rest | A))
   (~ k in A ==> [k v] ++ rest | = 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 | = empty-map)
          ==> (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 m | = 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 m | = A) [restrict-axioms]
              ==> (x in A) [Set.SC]])
          )
        )
      | (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) [range-characterization]]
  <==> 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 ˆ| 
[alist->fmap Set.lst->set]
]

assert* range-restricted-def :=

(empty-map ˆ| _ = empty-map)

((k v) ++ rest ˆ| A = [k v] ++ (rest - k ˆ| A) <= v in A)

((k v) ++ rest ˆ| A = rest - k ˆ| A <= ~ v in A)]

(define p (forall m A . range m ˆ| A subset range m))

define eye-color :=
[['bob --> 'brown'] ['tom --> 'blue'] ['lisa --> 'green'] ['peter --> 'blue'] ['ann --> 'brown]]

(eval eye-color ˆ| ['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 ˆ| A subset range m)

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

(assert* agree-on-def :=

([agree-on m1 m2 A) <==> m1 |ˆ A = m2 |ˆ A])

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

(eval {agree-on ['a --> 1] ['b --> 2]}

['b --> 3] ['a --> 1]

['b]))

define 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 {[l --> 'a] [2 --> 'b] ** {[l --> 'foo] [3 --> 'c]})

conclude override-theorem-1 := (forall m . [] ** m = m)

by-induction override-theorem-1 {
(m as empty-map) =>

(let [IH := (!chain (empty-map ** m) = empty-map [override-def])]

(| (m as (update (pair k v) rest)) =>

(let [IH := (!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))]}
```plaintext
lib/basic/fmaps.ath

762 pick-any m2 : (Map 'K 'V)
763 { !chain ([dom m2 ++ m1])
764   = ([dom [k v] ++ (m2 ++ m1)]) [override-def]           
765   = ([k ++ dom (m2 ++ m1)]) [dom-axioms]
766   = ([k ++ dom m2) \ (dom rest)]) [IH]
767   = ((dom m2) \ k ++ dom rest) [Set.union-lemma-2]
768   = ((dom m2) \ dom m1) [dom-axioms]}
769
770 define conj2 :=
771 { forall m1 m2 k . k in dom m1 ==> (m2 ++ m1) applied-to k = m1 applied-to k }
772
773 # (falsify conj2 20)
774
776 by-induction conj2 {  
777   (m1 as empty-map : (Map 'S 'T)) =>
778   pick-any m2 : (Map 'S 'T) k : S
779    { !chain ([k in dom m1])
780      ==> (k in null) [dom-axioms]
781      ==> false [Set.NC]
782      ==> ((m2 ++ m1) applied-to k = m1 applied-to k) [prop-taut]}
783   | (m1 as update (pair key val) rest) =>
784     let {IH := (forall m2 k . k in dom rest ==> (m2 ++ rest) applied-to k = rest applied-to k) }
785     pick-any m2 k
786      assume hyp := (k in dom m1)
787      { !cases (!chain-> [hyp]
788        ==> (k in key ++ dom rest) [dom-axioms]
789        ==> (k = key | k in dom rest) [Set.in-def]
790        ==> (k = key | k /= key & k in dom rest) [prop-taut])}
791    assume (k = key)
792    { !chain [[[m2 ++ m1] applied-to k]]
793      = ([[key val] ++ (m2 ++ m1) applied-to k]) [override-def]
794      = ([[key val] ++ (m2 ++ m1) applied-to key]) [k = key]
795      = ([some val]) [apply-axioms]
796      = ([m1 applied-to k]) [apply-axioms]
797      = (m1 applied-to k) [dom-axioms]}
798    assume (k =/= key & k in dom rest)
799    { !chain [[[m2 ++ m1] applied-to k]]
800      = ([[key val] ++ (m2 ++ m1) applied-to k]) [override-def]
801      = ([[key val] ++ (m2 ++ m1) applied-to key]) [k = key]
802      = ([rest applied-to k]) [IH]
803      = (m1 applied-to k) [apply-axioms]}
804 }
805
806 define conj3 :=
807 { forall m1 m2 . range m2 ++ m1 = (range m2) \ (range m1) }
808 { falsify conj3 10 }
809 { falsify conj3 20 }
810
811 conclude restrict-theorem-3 :=
812 { forall m2 m1 A . (m1 ++ m2) \ A = m1 \ A ++ m2 \ A }
813
814 by-induction restrict-theorem-3 {  
815   (m2 as empty-map) =>
816   pick-any m1 A
817    { !combine-equations
818      { !chain [[[m1 ++ m2] \ A] = (m1 \ A)]}
819      { !chain [[[m1 \ A ++ m2 \ A] A]
820        = (m1 \ A ++ empty-map)
821        = (m1 \ A)]}
822    | (m2 as update (pair k v) rest) =>
823      let {IH := (forall m1 A . (m1 ++ rest) \ A = m1 \ A ++ rest \ A) }
824      pick-any m1 A
825      { !two-cases
826        assume k in A
827        { !combine-equations
828          { !chain [[[m1 ++ m2] \ A]
829            = (((k v) ++ (m1 ++ rest) \ A) [override-def]
830            = ((k v) ++ (m1 ++ rest) \ A) [restrict-axioms]
831            = ((k v) ++ (m1 \ A ++ rest \ A) [IH])
832          !chain [[[m1 \ A ++ m2 \ A]
833            = ((m1 \ A ++ m2 \ A)]}
834        }
835      }
836    }
837  }
838  }
839```
\[
\begin{align*}
&= (m1 \mid^* A \mid^* k v \mid^* \text{rest} \mid^* A) \ [\text{restrict-axioms}] \\
&= (k v \mid^* (m1 \mid^* A \mid^* \text{rest} \mid^* A)) \ [\text{override-def}])
\end{align*}
\]

\textbf{assert} \ (* k \ in \ A) \\
\textbf{!chain} \ (((m1 \mid^* m2) \mid^* A) \\
\begin{align*}
&= (((k v) \mid^* (m1 \mid^* \text{rest})) \mid^* A) \ [\text{override-def}] \\
&= ((m1 \mid^* \text{rest}) \mid^* A) \ [\text{restrict-axioms}] \\
&= (m1 \mid^* A \mid^* \text{rest} \mid^* A) \ [\text{IH}] \\
&= (m1 \mid^* A \mid^* m2 \mid^* A) \ [\text{restrict-axioms}])
\end{align*}
\]

\textbf{declare} compose: (S1, S2, S3) \rightarrow (Map S1 S2) \ [\text{o [alist->fmap alist->fmap]}]

\textbf{assert*} \ compose-def := \\
\begin{align*}
&\{(_\circ \emptyset = \emptyset) \\
&(m \circ (k v) \mid^* \text{rest} = k v \mid^* \text{rest}) \iff m \text { applied-to } v = \text{SOME } v' \\
&(m \circ (k v) \mid^* \text{rest} = m \mid^* \text{rest}) \iff m \text { applied-to } v = \text{NONE}
\end{align*}

\begin{align*}
&\text{(define M1 [[1 \rightarrow 'a] [2 \rightarrow 'b] [1 \rightarrow 'c]])} \\
&\text{(define M2 [['a \rightarrow true] ['b \rightarrow false] ['foo \rightarrow true]])} \\
&\text{eval M2 \mid M1}
\end{align*}

\textbf{define} capitals := \\
\begin{align*}
&[['\text{paris} \rightarrow '\text{france}] ['\text{tokyo} \rightarrow '\text{japan}] ['\text{cairo} \rightarrow '\text{egypt}]]
\end{align*}

\textbf{define} countries := \\
\begin{align*}
&[['\text{france} \rightarrow '\text{europe}] ['\text{algeria} \rightarrow '\text{africa}] ['\text{japan} \rightarrow '\text{asia}]]
\end{align*}

\textbf{eval} countries o capitals

\textbf{define} [t1 t2] \rightarrow (Map S1 S2) \ [\text{alist->fmap alist->fmap}]

\textbf{#(c t1 t2)}

\textbf{define} composition-is-comm := (forall m1 m2 . m1 \circ m2 = m2 \circ m1)

\textbf{#(falsify composition-is-comm 20)}

\textbf{define} composition-is-assoc := (forall m1 m2 m3 . m1 \circ (m2 \circ m3) = (m1 \circ m2) \circ m3)

\textbf{#(falsify composition-is-assoc 20)}

\textbf{define} [n] := \[?n:N\]

\textbf{declare} iterate: (S, S) \rightarrow (Map S S) \ [\text{alist->fmap int->nat}]

\textbf{define} ["iterated"] := \text{iterate iterate}

\textbf{assert*} \ iterate-axioms := \\
\begin{align*}
&\{m \circ^* \text{zero} = m \\
&(m \circ^* \text{suc} n = m \circ (m \circ^* n))
\end{align*}

\textbf{let} \ (m := (\text{alist->fmap} [[1 \rightarrow 2] [2 \rightarrow 3] [3 \rightarrow 1]])); \\
\_ := (\text{print \"m \text{iterated once: \"} (eval map->set m \circ^* 1)); \\
\_ := (\text{print \"m \text{iterated twice: \"} (eval map->set m \circ^* 2)); \\
\_ := (\text{print \"m \text{iterated thrice: \"} (eval map->set m \circ^* 3))

\textbf{assert*} \ compose2-def := \\
\begin{align*}
&(m \text{ compose2 empty-map = m}) \\
&(m \text{ compose2 } [k v] \mid^* \text{rest} = [k v'] \mid^* \text{rest} \iff m \text { applied-to } v = \text{SOME } v' \\
&(m \text{ compose2 } [k v] \mid^* \text{rest} = [k v] \mid^* \text{rest} \iff m \text { applied-to } v = \text{NONE})
\end{align*}

\textbf{define} compose2: (S) \rightarrow (Map S S) \ [\text{alist->fmap alist->fmap}]

\textbf{assert*} \ compose2-2-def := \\
\begin{align*}
&(m \text{ compose2 empty-map = m}) \\
&(m \text{ compose2 } [k v] \mid^* \text{rest} = [k v'] \mid^* \text{rest} \iff m \text { applied-to } v = \text{SOME } v' \\
&(m \text{ compose2 } [k v] \mid^* \text{rest} = [k v] \mid^* \text{rest} \iff m \text { applied-to } v = \text{NONE})
\end{align*}

\textbf{define} compose2-is-comm := (forall m1 m2 \cdot m1 \text{ compose2 m2 = m2 \text{ compose2 m1})}

\textbf{assert} compose2-is-comm := (forall m1 m2 \cdot m1 \text{ compose2 m2 = m2 \text{ compose2 m1})}

\textbf{assert} compose2-is-assoc := (forall m1 m2 m3 \cdot (m1 \text{ compose2 m2} o m3 = (m1 o m2) o m3)

\textbf{#(falsify compose2-is-assoc 20)}

\textbf{define} [n] := \[?n:N\]

\textbf{declare} iterate: (S, S) \rightarrow (Map S S) \ [\text{alist->fmap int->nat}]

\textbf{define} ["iterated"] := \text{iterate iterate}

\textbf{assert*} iterate-axioms := \\
\begin{align*}
&(m \circ^* \text{zero} = m) \\
&(m \circ^* \text{suc} n = m \circ (m \circ^* n))
\end{align*}

\textbf{let} \ (m := (\text{alist->fmap} [[1 \rightarrow 2] [2 \rightarrow 3] [3 \rightarrow 1]])); \\
\_ := (\text{print \"m \text{iterated once: \"} (eval map->set m \circ^* 1)); \\
\_ := (\text{print \"m \text{iterated twice: \"} (eval map->set m \circ^* 2)); \\
\_ := (\text{print \"m \text{iterated thrice: \"} (eval map->set m \circ^* 3))

\textbf{print \"m and m\text{\textendquote} ident\textendquote as\textendquote?)" (eval m = m \circ^* 3))

\textbf{declare} compose2: (S) \rightarrow (Map S S) \ [\text{alist->fmap alist->fmap}]

\textbf{assert*} \ compose2-2-def := \\
\begin{align*}
&(m \text{ compose2 empty-map = m}) \\
&(m \text{ compose2 } [k v] \mid^* \text{rest} = [k v'] \mid^* \text{rest} \iff m \text { applied-to } v = \text{SOME } v' \\
&(m \text{ compose2 } [k v] \mid^* \text{rest} = [k v] \mid^* \text{rest} \iff m \text { applied-to } v = \text{NONE})
\end{align*}

\textbf{define} compose2-is-comm := (forall m1 m2 \cdot m1 \text{ compose2 m2 = m2 \text{ compose2 m1})}

\textbf{assert} compose2-is-comm := (forall m1 m2 \cdot m1 \text{ compose2 m2 = m2 \text{ compose2 m1})}

\textbf{assert} compose2-is-assoc := (forall m1 m2 m3 \cdot (m1 \text{ compose2 m2} o m3 = (m1 o m2) o m3)

\textbf{#(falsify compose2-is-assoc 20)}
(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 evsll: 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"