

## lib/basic/fmaps.ath

```

1 load "sets"
2
3 load "strong-induction"
4
5 module FMap {
6
7 define succ := (string->symbol "S")
8 define < := N.<
9
10 define [A B C] := [?A:(Set.Set 'S1) ?B:(Set.Set 'S2) ?C:(Set.Set 'S3)]
11
12 structure (Map S T) := empty-map
13                       | (update (Pair S T) (Map S T))
14
15 assert (structure-axioms "Map")
16 define Pair := pair
17
18 define (alist->fmap-general L predecessor) :=
19   match L {
20     [] => empty-map
21     | (list-of (|| [x --> n] [x n]) rest) =>
22       (update (pair (preprocessor x) (preprocessor n)) (alist->fmap-general rest predecessor))
23     | _ => L
24   }
25
26 define (alist->fmap L) := (alist->fmap-general L id)
27
28 define (fmap->alist-general m predecessor) :=
29   match m {
30     empty-map => []
31     | (update (pair k v) rest) => (add [(preprocessor k) --> (preprocessor v)]
32                                       (fmap->alist-general rest predecessor))
33     | _ => m
34   }
35
36 define (fmap->alist m) := (fmap->alist-general m id)
37
38 define map-induction :=
39   method (goal premises)
40     match goal {
41       (forall (some-var x) (some-sentence body)) =>
42         let {property := lambda (m) (replace-var x m body)}
43           by-induction goal {
44             empty-map => (!vpf (property empty-map) premises)
45             | (update p m) =>
46               let {goal := (replace-var x (update p m) body);
47                   IH := (property m)}
48                 (!vpf goal (add IH premises))
49           }
50     }
51
52 define map-induction' :=
53   method (goal)
54     (!map-induction goal (ab))
55
56 define (alist->pair inner-1 inner-2) :=
57   lambda (L)
58     match L {
59       [a b] => ((inner-1 a) @ (inner-2 b))
60       | [a --> b] => ((inner-1 a) @ (inner-2 b))
61       | _ => L
62     }
63
64 expand-input update [(alist->pair id id) alist->fmap]
65
66 define :: := Cons
67

```

```

68 define [null ++ in subset proper-subset \/ /\ \ - card] :=
69   [Set.null Set.++ Set.in Set.subset Set.proper-subset
70     Set.\ Set./\ Set.\ Set.- Set.card]
71
72 overload ++ update
73 #set-precedence ++ 210
74
75
76
77 define [key key1 key2 k k' k1 k2] := [?key ?key1 ?key2 ?k ?k' ?k1 ?k2]
78 define [val val1 val2 v v' v1 v2 x x1 x2 y y1 y2] :=
79   [?val ?val1 ?val2 ?v ?v' ?v1 ?v2 ?x ?x1 ?x2 ?y ?y1 ?y2]
80 define [m m' m1 m2 m3 rest rest1] := [?m:(Map 'S1 'S2) ?m':(Map 'S1 'S2) ?m1:(Map 'S3 'S4)
81                                       ?m2:(Map 'S5 'S6) ?m3:(Map 'S7 'S8) ?rest:(Map 'S9 'S10) ?rest1:(Map 'S11 'S12)]
82 define [S S1 S2 S3] := [?S:(Set.Set 'S) ?S1:(Set.Set 'S1) ?S2:(Set.Set 'S2) ?S3:(Set.Set 'S3)]
83
84 define [L L1 L2 more more1] := [?L ?L1 ?L2 ?more ?more1]
85
86 declare apply: (S, T) [(Map S T) S] -> (Option T) [applied-to 110 [alist->fmap id]]
87
88 define at := applied-to
89
90 declare remove: (S, T) [(Map S T) S] -> (Map S T) [- 120 [alist->fmap id]]
91
92 left-assoc -
93
94 (define t1 (- ?x ?y))
95
96 define (removed-from key map) := (remove map key)
97
98 #assert* remove-axioms :=
99 # [(_ removed-from empty-map = empty-map)
100 # (key removed-from [key _] ++ rest = key removed-from rest)
101 # (key /= x ==> x removed-from [key val] ++ rest = [key val] ++ (x removed-from rest))]
102
103 assert* remove-def :=
104   [([] - _ = empty-map)
105     ([key _] ++ rest - key = rest - key)
106     (key /= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]
107
108 (define t2 (- ?x null))
109 (define t3 (- ?x ?y))
110
111 #define (- map key) := (key removed-from map)
112
113
114
115 define M := [[1 --> 'a] [2 --> 'b] [1 --> 'c]]
116 define ide-map := [['a --> 1] ['b --> 2] ['c --> 3] ['a --> 99]]
117 define ide-map' := [['b --> 2] ['c --> 3] ['a --> 1] ['a --> 99]]
118 define ide-map'' := [['b --> 2] ['c --> 3] ['a --> 1] ['d --> 4] ['a --> 99]]
119
120  #(set-flag mlstyle-fundef "on")
121
122 assert* apply-axioms :=
123   [([] at _ = NONE)
124     ([key val] ++ _ at x = SOME val <== key = x)
125     ([key _] ++ rest at x = rest at x <== key /= x)]
126
127 #define applied-to := apply
128 ## The following gives the result NONE:(Option 'T286327), but it should be NONE:(Option Int)
129 # (set-flag mlstyle-fundef "on")
130  #(apply' empty-map:(Map Int Int) 1) [FIXED]
131
132 (eval (empty-map:(Map Int Int) at 1))
133
134 (eval M applied-to 1)
135 (eval M applied-to 2)
136 (eval M applied-to 97)
137 (eval M - 1 applied-to 2)

```

```

138 (eval M - 1 applied-to 1)
139
140 conclude apply-lemma-1 :=
141   (forall key val rest x .
142     [key val] ++ rest at x = NONE ==> rest at x = NONE)
143   pick-any key val rest x
144     let {m := ([key val] ++ rest);
145         hyp := (m at x = NONE);
146         goal := (rest at x = NONE)}
147     assume hyp
148       (!two-cases
149         (!chain [(key = x)
150                 ==> (m at x = SOME val) [apply-axioms]
151                 ==> (m at x != NONE) [option-results]
152                 ==> (hyp & ~hyp) [augment]
153                 ==> goal [prop-taut]])
154         (!chain [(key != x)
155                 ==> (m at x = rest at x) [apply-axioms]
156                 ==> (NONE = rest at x) [hyp]
157                 ==> goal [sym]]))
158
159
160
161 conclude apply-lemma-2 :=
162   (forall k v rest x .
163     [k v] ++ rest applied-to x != NONE <==> k = x | rest applied-to x != NONE)
164   pick-any k v rest x
165     (!two-cases
166       assume case-1 := (k = x)
167         (!equiv assume hyp := ([k v] ++ rest applied-to x != NONE)
168           (!chain-> [(k = x) ==> (k = x | rest applied-to x != NONE) [alternate]])
169           assume (k = x | rest applied-to x != NONE)
170             (!chain-> [(k v] ++ rest applied-to x)
171               = ([x v] ++ rest applied-to x) [(k = x)]
172               = (SOME v) [apply-axioms]
173               ==> ([k v] ++ rest applied-to x != NONE) [option-results]))
174       assume case-2 := (k != x)
175         (!equiv assume hyp := ([k v] ++ rest applied-to x != NONE)
176           (!chain-> [hyp
177                     ==> (rest applied-to x != NONE) [apply-axioms]
178                     ==> (k = x | rest applied-to x != NONE) [alternate]])
179           assume C := (k = x | rest applied-to x != NONE)
180             (!cases C
181               assume (k = x)
182                 (!from-complements ([k v] ++ rest applied-to x != NONE) (k = x) (k != x))
183                 (!chain [(rest applied-to x != NONE) ==> ([k v] ++ rest applied-to x != NONE) [apply-lemma-1]]))
184
185 conclude apply-lemma-3 :=
186   (forall m k v1 v2 . m applied-to k = SOME v1 & m applied-to k = SOME v2 ==> v1 = v2)
187   pick-any m k v1 v2
188     assume hyp := (m applied-to k = SOME v1 & m applied-to k = SOME v2)
189     (!chain-> [(SOME v1)
190               = (m applied-to k)
191               = (SOME v2)
192               ==> (v1 = v2) [option-results]])
193
194 conclude remove-correctness :=
195   (forall m x . m - x applied-to x = NONE)
196   by-induction remove-correctness {
197     (m as empty-map) =>
198     pick-any x
199       (!chain [([] - x applied-to x)
200               = ([] applied-to x) [remove-def]
201               = NONE [apply-axioms]])
202   | (m as (update (pair key val) rest)) =>
203     let {IH := (forall x . rest - x applied-to x = NONE)}
204     pick-any x
205       (!two-cases
206         assume case1 := (key = x)
207         (!chain [(m - x applied-to x)

```

```

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
218 define (RC2-M goal p1 p2) :=
219   match [goal p1 p2] {
220     [(~ (s = t)) (s = u) (~ (u = t))] =>
221       (!by-contradiction goal
222         assume (~ goal)
223         (!chain-> [(~ goal)
224                 ==> (s = t) [dn]
225                 ==> (u = t) [(s = u)]
226                 ==> (u = t & u != t) [augment]
227                 ==> false [prop-taut]]))
228   }
229
230
231 conclude remove-correctness-2 :=
232   (forall m x y . x != y ==> (m - x) at y = m at y)
233 by-induction remove-correctness-2 {
234   (m as empty-map) =>
235     pick-any x y
236     assume hyp := (x != y)
237     (!chain [(m - x) at y
238             = (m at y) [remove-def]])
239 | (m as (update (pair key val) rest)) =>
240   let {IH := (forall x y . x != y ==> (rest - x) at y = rest at y)}
241   pick-any x y
242   assume hyp := (x != y)
243   (!two-cases
244     assume case1 := (key = x)
245     #let {lemma := (!CongruenceClosure.cc (key != y) [case1 hyp])}
246     let {lemma := (!RC2-M (key != y) case1 hyp)}
247     (!chain [(m - x) at y
248             = ((rest - x) at y) [(key = x) remove-def]
249             = (rest at y) [IH]
250             = (m at y) [apply-axioms]])
251     assume (key != x)
252     (!two-cases
253       assume (key = y)
254       (!combine-equations
255         (!chain [(m - x) at y
256                 = (([key val] ++ (rest - x)) at y) [remove-def]
257                 = (SOME val) [apply-axioms]])
258         (!chain [(m at y)
259                 = (SOME val) [apply-axioms]])
260         assume (key != y)
261         (!combine-equations
262           (!chain [(m - x) at y
263                   = (([key val] ++ (rest - x)) at y) [remove-def]
264                   = ((rest - x) at y) [apply-axioms]
265                   = (rest at y) [IH]])
266           (!chain [(m at y)
267                   = (rest at y) [apply-axioms]]))
268   }
269
270 declare map->set: (S, T) [(Map S T)] -> (Set.Set (Pair S T)) [[alist->fmap]]
271
272 assert* map->set-def :=
273   [(map->set empty-map = null)
274    (map->set [k v] ++ rest = (k @ v) ++ map->set rest - k)]
275
276 assert* map-identity := (m1 = m2 <==> map->set m1 = map->set m2)
277

```

```

278 (eval map->set ide-map)
279 (eval (alist->fmap ide-map) = (alist->fmap ide-map'))
280 (eval (alist->fmap ide-map) = (alist->fmap ide-map"))
281
282
283 conclude opair-lemma :=
284   (forall x1 x2 y1 y2 A . x1 /= x2 ==> x1 @ y1 in A <==> x1 @ y1 in x2 @ y2 ++ A)
285 pick-any x1:'S x2:'S y1:'T y2:'T A:(Set.Set (Pair 'S 'T))
286   assume (x1 /= x2)
287     (!equiv (!chain [(x1 @ y1 in A)
288                       ==> (x1 @ y1 in x2 @ y2 ++ A)           [Set.in-lemma-3]]
289                       (!chain [(x1 @ y1 in x2 @ y2 ++ A)
290                                 ==> (x1 @ y1 = x2 @ y2 | x1 @ y1 in A)   [Set.in-def]
291                                 ==> ((x1 = x2 & y1 = y2) | x1 @ y1 in A) [(datatype-axioms "Pair")]
292                                 ==> (x1 = x2 | x1 @ y1 in A)           [prop-taut]
293                                 ==> (x1 /= x2 & (x1 = x2 | x1 @ y1 in A)) [augment]
294                                 ==> ((x1 /= x2 & x1 = x2) | (x1 /= x2 & x1 @ y1 in A)) [prop-taut]
295                                 ==> (false | (x1 /= x2 & x1 @ y1 in A))     [prop-taut]
296                                 ==> (x1 /= x2 & x1 @ y1 in A)           [prop-taut]
297                                 ==> (x1 @ y1 in A)                       [right-and]]))
298
299
300
301 define ms-lemma-1a :=
302 pick-any x key val rest v
303   assume hyp := (x /= key)
304     (!chain [(key _) ++ rest applied-to x = SOME v)
305             <==> (rest applied-to x = SOME v) [apply-axioms]])
306
307 #define ms-lemma-1b :=
308 # (forall m k v x . k @ v in map->set m & x /= k ==> k @ v in map->set (m - x))
309
310 declare dom: (S, T) [(Map S T)] -> (Set.Set S) [[alist->fmap]]
311
312 assert* dom-axioms :=
313   [(dom empty-map = null)
314    (dom [k _] ++ rest = k ++ dom rest)]
315
316 transform-output eval [Set.set->lst fmap->alist]
317
318 (eval dom ide-map)
319
320 conclude dom-lemma-1 :=
321   (forall k v rest . k in dom [k v] ++ rest)
322 pick-any k v rest
323   (!chain-> [true ==> (k in k ++ dom rest)           [Set.in-lemma-1]
324             ==> (k in dom [k v] ++ rest) [dom-axioms]])
325
326 conclude dom-lemma-2 :=
327   (forall m k v . dom m subset dom [k v] ++ m)
328 pick-any m k v
329   (!Set.subset-intro
330     pick-any x
331       (!chain [(x in dom m)
332                 ==> (x in k ++ dom m)           [Set.in-lemma-3]
333                 ==> (x in dom [k v] ++ m)     [dom-axioms]]))
334
335
336 conclude dom-characterization :=
337   (forall m k . k in dom m <==> m applied-to k /= NONE)
338 by-induction dom-characterization {
339   (m as empty-map) =>
340     pick-any k
341       (!equiv
342         (!chain [(k in dom m)
343                   ==> (k in null)           [dom-axioms]
344                   ==> false                 [Set.NC]
345                   ==> (m applied-to k /= NONE) [prop-taut]])
346         assume hyp := (m applied-to k /= NONE)
347         (!chain-> [true

```

```

348     ==> (m applied-to k = NONE)           [apply-axioms]
349     ==> (m applied-to k = NONE & hyp)     [augment]
350     ==> false                             [prop-taut]
351     ==> (k in dom m)                       [prop-taut]))
352 | (m as (update (pair x y) rest)) =>
353   let {IH := (forall k . k in dom rest <==> rest applied-to k /= NONE)}
354   pick-any k
355     (!chain [(k in dom m)
356             <==> (k in x ++ dom rest)       [dom-axioms]
357             <==> (k = x | k in dom rest)     [Set.in-def]
358             <==> (k = x | rest applied-to k /= NONE) [IH]
359             <==> (x = k | rest applied-to k /= NONE) [sym]
360             <==> (m applied-to k /= NONE)     [apply-lemma-2]])
361 }
362
363 conclude dom-lemma-3 := (forall m k . dom (m - k) subset dom m)
364 by-induction dom-lemma-3 {
365   (m as empty-map:(Map 'K 'V)) =>
366     pick-any k:'K
367       (!Set.subset-intro
368         pick-any x:'K
369           (!chain [(x in dom m - k)
370                 ==> (x in dom empty-map)   [remove-def]
371                 ==> (x in null)           [dom-axioms]
372                 ==> false                 [Set.NC]
373                 ==> (x in dom m)         [prop-taut]]))
374 | (m as (update (pair key:'K val:'V) rest)) =>
375   pick-any k:'K
376     let {IH := (!claim (forall k . dom rest - k subset dom rest));
377         IH1 := (!chain-> [true ==> (dom rest - key subset dom rest) [IH]]);
378         IH2 := (!chain-> [true ==> (dom rest - k subset dom rest) [IH]])}
379     (!Set.subset-intro
380       pick-any x:'K
381         (!two-cases
382           assume (key = k)
383             (!chain [(x in dom m - k)
384                   ==> (x in dom m - key)   [(key = k)]
385                   ==> (x in dom rest - key) [remove-def]
386                   ==> (x in dom rest)     [IH1 Set.SC]
387                   ==> (x in key ++ dom rest) [Set.in-lemma-3]
388                   ==> (x in dom m)       [dom-axioms]])
389           assume case-2 := (key /= k)
390             (!chain [(x in dom m - k)
391                   ==> (x in dom [key val] ++ (rest - k)) [remove-def]
392                   ==> (x in key ++ dom rest - k)       [dom-axioms]
393                   ==> (x = key | x in dom rest - k)     [Set.in-def]
394                   ==> (x = key | x in dom rest)         [Set.SC IH2]
395                   ==> (x in key ++ dom rest)           [Set.in-def]
396                   ==> (x in dom m)                     [dom-axioms]]))
397 }
398
399
400 declare size: (S, T) [(Map S T)] -> N [[alist->fmap]]
401
402 assert* size-axioms := [(size m = card dom m)]
403
404 transform-output eval [nat->int]
405
406 (eval size ide-map)
407
408 conclude ms-rec-lemma :=
409   (forall m k v . size (m - k) < size [k v] ++ m)
410
411 conclude ms-rec-lemma
412 pick-any m:(Map 'K 'V) key:'K val:'V
413   let {L1 := (!by-contradiction (~ key in dom m - key)
414         assume h := (key in dom m - key)
415           (!absurd (!chain-> [true ==> ((m - key) applied-to key = NONE) [remove-correctness]]
416             (!chain-> [h ==> ((m - key) applied-to key /= NONE) [dom-characterization]]))));
417       L2 := (!chain-> [true ==> (key in dom [key val] ++ m) [dom-lemma-1]]);

```

```

418     L3 := (!both (!chain-> [true ==> (dom m - key subset dom m) [dom-lemma-3]])
419           (!chain-> [true ==> (dom m subset dom [key val] ++ m) [dom-lemma-2]]));
420     L4 := (!chain-> [L3 ==> (dom m - key subset dom [key val] ++ m) [Set.subset-transitivity]]))
421     (!chain-> [L4 ==> (L4 & L2 & L1) [augment]
422              ==> (dom m - key proper-subset dom [key val] ++ m) [Set.proper-subset-lemma]
423              ==> (card dom m - key < card dom [key val] ++ m) [Set.proper-subset-card-theorem]
424              ==> (size m - key < size [key val] ++ m) [size-axioms]))
425
426 define ms-theorem :=
427   (forall m k v . k @ v in map->set m <==> m applied-to k = SOME v)
428
429 (define (property m)
430   (forall k v . k @ v in map->set m <==> m applied-to k = SOME v))
431
432 conclude ms-theorem
433   (!strong-induction.measure-induction ms-theorem size
434     pick-any m:(Map 'K 'V)
435     assume IH := (forall m' . size m' < size m ==> property m')
436     conclude (property m)
437     datatype-cases (property m) on m {
438       (em as empty-map:(Map 'K 'V)) =>
439       (pick-any k:'K v:'V
440        let {none := NONE:(Option 'V)}
441          (!equiv (!chain [(k @ v in map->set em)
442                          ==> (k @ v in null)
443                          ==> false
444                          ==> (em applied-to k = SOME v)]))
445          assume hyp := (em applied-to k = SOME v)
446          (!chain-> [true
447                   ==> (em applied-to k = none) [apply-axioms]
448                   ==> (em applied-to k = none & hyp) [augment]
449                   ==> (em applied-to k = none & em applied-to k /= none) [option-results]
450                   ==> false [prop-taut]
451                   ==> (k @ v in map->set em) [prop-taut]]))
452 | (map as (update (pair key:'K val:'V) rest)) =>
453   pick-any k:'K v:'V
454   let {goal := (k @ v in map->set map <==> map applied-to k = SOME v);
455        lemma := (!chain-> [true ==> (size rest - key < size map) [ms-rec-lemma]
456                             ==> (size rest - key < size m) [(m = map)]])}
457   (!two-cases
458     assume casel := (k = key)
459     (!equiv assume hyp := (k @ v in map->set map)
460      let {D := (!chain-> [hyp
461                          ==> (k @ v in key @ val ++ map->set rest - key)
462                          ==> (key @ v in key @ val ++ map->set rest - key) [casel]
463                          ==> (key @ v = key @ val | key @ v in map->set rest - key) [Set.in-de]
464                          (!cases D
465                            assume h1 := (key @ v in map->set rest - key)
466                            let {_ := (!absurd (!chain-> [h1 ==> ((rest - key) applied-to key = SOME v)
467                                                    ==> (NONE = SOME v)
468                                                    ==> (NONE /= SOME v) [option-results]])]}
469                            (!from-false (map applied-to k = SOME v))
470                            assume h2 := (key @ v = key @ val)
471                            let {v=val := (!chain [h2 ==> (v = val)])}
472                            (!chain-> [(map applied-to key) = (SOME val) [apply-axioms]
473                                       = (SOME v) [v=val]
474                                       ==> (map applied-to key = SOME v)
475                                       ==> (map applied-to k = SOME v) [(k = key)]])}
476                            assume hyp := (map applied-to k = SOME v)
477                            let {val=v := (!chain-> [(SOME val)
478                                                    = (map applied-to key) [apply-axioms]
479                                                    = (map applied-to k) [(k = key)]
480                                                    = (SOME v) [hyp]
481                                                    ==> (val = v) [option-results]])}
482                            (!chain-> [true ==> (key @ val in key @ val ++ map->set (rest - key)) [Set.in-lemma-
483                                       ==> (key @ val in map->set map)

```

```

484                                     ==> (k @ val in map->set map)
485 [ (k = key) ]
486                                     ==> (k @ v in map->set map)
487 [val=v]])
488
489     assume case2 := (k != key)
490     (!iff-comm
491       (!chain [(map applied-to k = SOME v)
492         <==> (rest applied-to k = SOME v)
493           [apply-axioms]
494         <==> ((rest - key) applied-to k = SOME v)
495           [remove-correctness-2]
496         <==> (k @ v in map->set rest - key)
497           [IH]
498         <==> (k @ v in key @ val ++ map->set rest - key) [(k @ v in map->set rest - key <==>
499           k @ v in key @ val ++ map->set rest - key)
500           <== case2 [opair-lemma]]
501         <==> (k @ v in map->set map)
502           [map->set-def]]))
503   ))
504 (eval dom ide-map)
505
506 conclude dom-characterization-2 :=
507 (forall m x . x in dom m <==> exists v . x @ v in map->set m)
508 pick-any m:(Map 'K 'V) x:'K
509 (!chain [(x in dom m)
510   <==> (m applied-to x != NONE)
511     [dom-characterization]
512   <==> (exists v . m applied-to x = SOME v)
513     [option-results]
514   <==> (exists v . x @ v in map->set m)
515     [ms-theorem]])
516
517 conclude ms-corollary :=
518 (forall m k . m applied-to k = NONE <==> ~ exists v . k @ v in map->set m)
519 pick-any m:(Map 'K 'V) k:'K
520 (!equiv (!chain [(m applied-to k = NONE)
521   ==> (~ exists v . m applied-to k = SOME v)
522     [option-results]
523   ==> (~ exists v . k @ v in map->set m)
524     [ms-theorem]])
525 (!chain [(~ exists v . k @ v in map->set m)
526   ==> (~ exists v . m applied-to k = SOME v)
527     [ms-theorem]
528   ==> (m applied-to k = NONE)
529     [option-results]]))
530
531 conclude identity-characterization-1 :=
532 (forall m1 m2 . m1 = m2 ==> forall k . m1 applied-to k = m2 applied-to k)
533 pick-any m1:(Map 'S 'T) m2:(Map 'S 'T)
534 assume hyp := (m1 = m2)
535 let {m1=m2 := (!chain-> [hyp ==> (map->set m1 = map->set m2) [map-identity]])}
536 pick-any k:'S
537 (!cases (!chain-> [true ==> (m1 applied-to k = NONE | exists v . m1 applied-to k = SOME v) [option-results]])
538   assume case1 := (m1 applied-to k = NONE)
539     let {p := (!by-contradiction (m2 applied-to k = NONE)
540       assume h := (m2 applied-to k != NONE)
541         pick-witness v for (!chain-> [h ==> (exists v . m2 applied-to k = SOME v) [option-results]])
542           (!chain-> [wp ==> (k @ v in map->set m2) [ms-theorem]
543             ==> (k @ v in map->set m1) [m1=m2]
544             ==> (m1 applied-to k = SOME v) [ms-theorem]
545             ==> (m1 applied-to k != NONE) [option-results]
546             ==> (case1 & m1 applied-to k != NONE) [augment]
547             ==> false [prop-taut]])})
548     (!combine-equations (m1 applied-to k = NONE) (m2 applied-to k = NONE))
549     assume case2 := (exists v . m1 applied-to k = SOME v)
550       pick-witness v for case2
551         (!combine-equations
552           (m1 applied-to k = SOME v)
553           (!chain-> [(m1 applied-to k = SOME v)
554             ==> (k @ v in map->set m1) [ms-theorem]
555             ==> (k @ v in map->set m2) [m1=m2]
556             ==> (m2 applied-to k = SOME v) [ms-theorem]]))
557
558 conclude identity-characterization-2 :=
559 (forall m1 m2 . (forall k . m1 applied-to k = m2 applied-to k) ==> m1 = m2)
560 pick-any m1:(Map 'S 'T) m2:(Map 'S 'T)
561 assume hyp := (forall k . m1 applied-to k = m2 applied-to k)
562 let {m1=m2-as-sets :=
563   (!Set.set-identity-intro-direct

```



```

552     (!pair-converter
553       pick-any k:'S v:'T
554       (!chain [(k @ v in map->set m1)
555         <==> (m1 applied-to k = SOME v)           [ms-theorem]
556         <==> (m2 applied-to k = SOME v)           [hyp]
557         <==> (k @ v in map->set m2)           [ms-theorem]]]))
558     (!chain-> [m1=m2-as-sets ==> (m1 = m2) [map-identity]])
559
560 conclude identity-characterization :=
561   (forall m1 m2 . m1 = m2 <==> forall k . m1 applied-to k = m2 applied-to k)
562 pick-any m1:(Map 'S 'T) m2:(Map 'S 'T)
563   (!equiv
564     (!chain [(m1 = m2) ==> (forall k . m1 applied-to k = m2 applied-to k) [identity-characterization-1]])
565     (!chain [(forall k . m1 applied-to k = m2 applied-to k) ==> (m1 = m2) [identity-characterization-2]]))
566
567
568 declare restricted-to: (S, T) [(Map S T) (Set.Set S)] -> (Map S T) [150 |^ [alist->fmap Set.lst->set]]
569
570 assert* restrict-axioms :=
571   [(empty-map |^ _ = empty-map)
572    (k in A ==> [k v] ++ rest |^ A = [k v] ++ (rest |^ A))
573    (~ k in A ==> [k v] ++ rest |^ A = rest |^ A)]
574
575 (eval [[1 --> 'a] [2 --> 'b] [3 --> 'c]] |^ [1 3])
576
577 conclude restriction-theorem-1 := (forall m A . dom m |^ A subset A)
578 by-induction restriction-theorem-1 {
579   empty-map =>
580     pick-any A
581       (!Set.subset-intro
582         pick-any x
583           (!chain [(x in dom empty-map |^ A)
584             ==> (x in dom empty-map)           [restrict-axioms]
585             ==> (x in null)                   [dom-axioms]
586             ==> false                         [Set.NC]
587             ==> (x in A)                     [prop-taut]]))
588 | (m as (update (pair k v) rest)) =>
589   pick-any A
590   let {IH := (forall A . dom rest |^ A subset A);
591     lemma := (!chain-> [true ==> (dom rest |^ A subset A) [IH]])}
592   (!two-cases
593     assume case-1 := (k in A)
594     (!Set.subset-intro
595       pick-any x
596         (!chain [(x in dom m |^ A)
597           ==> (x in dom [k v] ++ (rest |^ A)) [restrict-axioms]
598           ==> (x in k ++ dom rest |^ A)       [dom-axioms]
599           ==> (x = k | x in dom rest |^ A)    [Set.in-def]
600           ==> (x in A | x in dom rest |^ A)  [case-1]
601           ==> (x in A | x in A)              [Set.SC]
602           ==> (x in A)                       [prop-taut]]))
603     assume case-2 := (~ k in A)
604     (!Set.subset-intro
605       pick-any x
606         (!chain [(x in dom m |^ A)
607           ==> (x in dom rest |^ A) [restrict-axioms]
608           ==> (x in A)           [Set.SC]]))
609   }
610
611
612 conclude restriction-theorem-2 :=
613   (forall m A . dom m subset A ==> m |^ A = m)
614 by-induction restriction-theorem-2 {
615   (m as empty-map) =>
616     pick-any A
617       assume hyp := (dom m subset A)
618       (!chain [(m |^ A) = m [restrict-axioms]])
619 | (m as (update (pair key val) rest)) =>
620   pick-any A
621   assume hyp := (dom m subset A)

```

```

622     let {lemma1 := (!chain-> [true ==> (key in dom m) [dom-lemma-1]
623                               ==> (key in A)      [Set.SC]]);
624         lemma2 := (!chain-> [true ==> (dom rest subset dom m)      [dom-lemma-2]
625                               ==> (dom rest subset dom m & hyp) [augment]
626                               ==> (dom rest subset A)           [Set.subset-transitivity]]);
627     IH := (forall A . dom rest subset A ==> rest |^ A = rest)
628     (!chain [(m |^ A)
629              = ([key val] ++ (rest |^ A)) [restrict-axioms]
630              = ([key val] ++ rest)       [IH]])
631 }
632
633
634 declare range: (S, T) [(Map S T)] -> (Set.Set T) [[alist->fmap]]
635
636 assert* range-def :=
637   [(range m = Set.range map->set m)]
638
639 (eval range ide-map)
640
641 conclude range-lemma-1 :=
642   (forall m v . v in range m <==> exists k . k @ v in map->set m)
643 pick-any m v
644   (!chain [(v in range m)
645            <==> (v in Set.range map->set m) [range-def]
646            <==> (exists k . k @ v in map->set m) [Set.range-characterization]])
647
648 conclude range-characterization :=
649   (forall m v . v in range m <==> exists k . m at k = SOME v)
650 pick-any m v
651   (!chain [(v in range m)
652            <==> (exists k . k @ v in map->set m) [range-lemma-1]
653            <==> (exists k . m at k = SOME v) [ms-theorem]])
654
655 conclude range-lemma-2 :=
656   (forall k v rest . v in range [k v] ++ rest)
657 pick-any k v rest
658   (!chain<- [(v in range [k v] ++ rest)
659             <== (v in Set.range map->set [k v] ++ rest) [range-def]
660             <== (v in Set.range k @ v ++ map->set rest - k) [map->set-def]
661             <== (v in v ++ Set.range map->set rest - k) [Set.range-def]
662             <== (v = v | v in Set.range map->set rest - k) [Set.in-def]
663             <== (v = v) [alternate]])
664
665 define range-lemma-conjecture :=
666   (forall m k v . range m subset range [k v] ++ m)
667
668 (falsify range-lemma-conjecture 10)
669
670 conclude removal-range-theorem :=
671   (forall m k . range m - k subset range m)
672 pick-any m k
673   (!Set.subset-intro
674     pick-any v
675     assume hyp := (in v range m - k)
676     pick-witness key for
677       (!chain<- [(exists key . m - k at key = SOME v)
678                 <== hyp [range-characterization]])
679     key-premise
680     let {k!=key :=
681           (!by-contradiction (k /= key)
682             assume (k = key)
683               (!absurd (!chain-> [key-premise
684                                   ==> (m - key at key = SOME v) [(k = key)]])
685                 (!chain-> [true ==> (m - key at key = NONE) [remove-correctness]
686                           ==> (m - key at key /= SOME v) [option-results]])))))
687     (!chain-> [k!=key ==> (m - k at key = m at key) [remove-correctness-2]
688              ==> (SOME v = m at key) [key-premise]
689              ==> (m at key = SOME v) [sym]
690              ==> (exists key . m at key = SOME v) [existence]
691              ==> (v in range m) [range-characterization]]))

```

```

692
693 declare range-restricted: (S, T) [(Map S T) (Set.Set T)] -> (Map S T) [150 ^| [alist->fmap Set.lst->set]]
694
695 assert* range-restricted-def :=
696   [(empty-map ^| _ = empty-map)
697    ([k v] ++ rest ^| A = [k v] ++ (rest - k ^| A) <== v in A)
698    ([k v] ++ rest ^| A = rest - k ^| A <== ~ v in A)]
699
700 (define p (forall m A . range m ^| A subset range m))
701 define eye-color := [[ 'bob --> 'brown] [ 'tom --> 'blue] [ 'lisa --> 'green]
702                    [ 'peter --> 'blue] [ 'ann --> 'brown]]
703
704 (eval eye-color ^| [ 'blue])
705
706 (define vpf
707   (method (goal premises)
708     (!vprove-from goal premises [[ 'poly true] [ 'subsorting false] [ 'max-time 3000]])))
709
710 (define spf
711   (method (goal premises)
712     (!sprove-from goal premises [[ 'poly true] [ 'subsorting false] [ 'max-time 300]])))
713
714 ### CAUTION: THE PATTERN (m as null) seemed to work!
715
716 define range-restriction-theorem-1 :=
717   (forall m A . range m ^| A subset range m)
718
719 declare agree-on: (S, T) [(Map S T) (Map S T) (Set.Set S)] -> Boolean
720                    [[alist->fmap alist->fmap Set.lst->set]]
721
722 assert* agree-on-def := [(agree-on m1 m2 A) <==> m1 |^ A = m2 |^ A]
723
724 (eval (agree-on ide-map ide-map [ 'a 'b]))
725
726 (eval (agree-on [[ 'a --> 1] [ 'b --> 2]]
727              [[ 'b --> 3] [ 'a --> 1]]
728              [ 'b]))
729
730 declare override: (S, T) [(Map S T) (Map S T)] -> (Map S T) [** [alist->fmap alist->fmap]]
731
732 assert* override-def :=
733   [(m ** [] = m)
734    (m ** [k v] ++ rest = [k v] ++ (m ** rest))]
735
736 (eval [[1 --> 'a] [2 --> 'b]] ** [[1 --> 'foo] [3 --> 'c]])
737
738
739 conclude override-theorem-1 := (forall m . [] ** m = m)
740 by-induction override-theorem-1 {
741   (m as empty-map) =>
742     (!chain [(empty-map ** m) = empty-map [override-def]])
743 | (m as (update (pair k v) rest)) =>
744   let {IH := ([] ** rest = rest)}
745     (!chain [(empty-map ** m)
746              = ([k v] ++ (empty-map ** rest)) [override-def]
747              = ([k v] ++ rest) [IH]])
748 }
749
750 define conj1 := (forall m1 m2 . dom m2 ** m1 = (dom m2) \ / (dom m1))
751
752 by-induction (forall m1 m2 . dom m2 ** m1 = (dom m2) \ / (dom m1)) {
753   (m1 as empty-map: (Map 'K 'V)) =>
754     pick-any m2: (Map 'K 'V)
755       (!chain [(dom m2 ** m1)
756                = (dom m2) [override-def]
757                = (null \ / dom m2) [Set.union-def]
758                = ((dom m2) \ / null) [Set.union-commutes]
759                = ((dom m2) \ / (dom m1)) [dom-axioms]])
760 | (m1 as (update (pair k: 'K v: 'V) rest)) =>
761   let {IH := (forall m2 . dom m2 ** rest = (dom m2) \ / (dom rest))}

```

```

762   pick-any m2:(Map 'K 'V)
763     (!chain [(dom m2 ** m1)
764             = (dom [k v] ++ (m2 ** rest))      [override-def]
765             = (k ++ dom (m2 ** rest))         [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
771 define conj2 :=
772   (forall m1 m2 k . k in dom m1 ==> (m2 ** m1) applied-to k = m1 applied-to k)
773
774
775 # (falsify conj2 20)
776
777 by-induction conj2 {
778   (m1 as empty-map:(Map 'S 'T)) =>
779     pick-any m2:(Map 'S 'T) k:'S
780       (!chain [(k in dom m1)
781               ==> (k in null)      [dom-axioms]
782               ==> false           [Set.NC]
783               ==> ((m2 ** m1) applied-to k = m1 applied-to k) [prop-taut]])
784 | (m1 as (update (pair key val) rest)) =>
785   let {IH := (forall m2 k . k in dom rest ==> (m2 ** rest) applied-to k = rest applied-to k)}
786     pick-any m2 k
787       assume hyp := (k in dom m1)
788         (!cases (!chain-> [hyp
789                          ==> (k in key ++ dom rest)      [dom-axioms]
790                          ==> (k = key | k in dom rest)   [Set.in-def]
791                          ==> (k = key | k /= key & k in dom rest) [prop-taut]])
792         assume (k = key)
793           (!chain [(m2 ** m1) applied-to k)
794                 = ([key val] ++ (m2 ** rest) applied-to k) [override-def]
795                 = ([key val] ++ (m2 ** rest) applied-to key) [(k = key)]
796                 = (SOME val)                                [apply-axioms]
797                 = (m1 applied-to key)                       [apply-axioms]
798                 = (m1 applied-to k)                         [(k = key)]]
799         assume (k /= key & k in dom rest)
800           (!chain [(m2 ** m1) applied-to k)
801                 = ([[key val] ++ (m2 ** rest)) applied-to k] [override-def]
802                 = ((m2 ** rest) applied-to k)                [apply-axioms]
803                 = (rest applied-to k)                        [IH]
804                 = (m1 applied-to k)                          [apply-axioms]])
805 }
806
807 define conj3 := (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 by-induction restrict-theorem-3 {
814   (m2 as empty-map) =>
815     pick-any m1 A
816       (!combine-equations
817         (!chain [(m1 ** m2) |^ A) = (m1 |^ A)])
818         (!chain [(m1 |^ A ** m2 |^ A)
819                 = (m1 |^ A ** empty-map)
820                 = (m1 |^ A)]))
821 | (m2 as (update (pair k v) rest)) =>
822   let {IH := (forall m1 A . (m1 ** rest) |^ A = m1 |^ A ** rest |^ A)}
823     pick-any m1 A
824       (!two-cases
825         assume (k in A)
826           (!combine-equations
827             (!chain [(m1 ** m2) |^ A)
828                   = ([[k v] ++ (m1 ** rest)) |^ A] [override-def]
829                   = ([k v] ++ (m1 ** rest) |^ A) [restrict-axioms]
830                   = ([k v] ++ (m1 |^ A ** rest |^ A)) [IH]])
831             (!chain [(m1 |^ A ** m2 |^ A)

```

```

832         = (m1 |^ A ** [k v] ++ (rest |^ A)) [restrict-axioms]
833         = ([k v] ++ (m1 |^ A ** rest |^ A)) [override-def]))
834     assume (~ k in A)
835         (!chain [(m1 ** m2) |^ A]
836             = ([k v] ++ (m1 ** rest)) |^ A)           [override-def]
837             = (m1 ** rest) |^ A)                     [restrict-axioms]
838             = (m1 |^ A ** rest |^ A)                  [IH]
839             = (m1 |^ A ** m2 |^ A)                    [restrict-axioms]))))
840 }
841
842 declare compose: (S1, S2, S3) [(Map S2 S3) (Map S1 S2)] -> (Map S1 S3) [o [alist->fmap alist->fmap]]
843
844 assert* compose-def :=
845     [(_ o empty-map = empty-map)
846      (m o [k v] ++ more = [k v'] ++ (m o more) <== m applied-to v = SOME v')
847      (m o [k v] ++ more = m o more <== m applied-to v = NONE)]
848
849 (define M1 [[1 --> 'a] [2 --> 'b] [1 --> 'c]])
850 (define M2 [['a --> true] ['b --> false] ['foo --> true]])
851 (eval M2 o M1)
852
853 define capitals :=
854     [['paris --> 'france] ['tokyo --> 'japan] ['cairo --> 'egypt]]
855
856 define countries :=
857     [['france --> 'europe] ['algeria --> 'africa] ['japan --> 'asia]]
858
859 (eval countries o capitals)
860
861 (define [t1 t2] [(alist->fmap M2) (alist->fmap M1)])
862  #(c t1 t2)
863
864 define composition-is-comm := (forall m1 m2 . m1 o m2 = m2 o m1)
865
866 (falsify composition-is-comm 20)
867
868 define composition-is-assoc := (forall m1 m2 m3 . m1 o (m2 o m3) = (m1 o m2) o m3)
869  #(falsify composition-is-assoc 20)
870
871  #(falsify (close ((m3 o m2) o m1) = m3 o (m2 o m1))) 20)
872
873 define [n] := [?n:N]
874
875 declare iterate: (S, S) [(Map S S) N] -> (Map S S) [^^ [alist->fmap int->nat]]
876 define [^^ iterated] := [iterate iterate]
877
878 assert* iterate-axioms :=
879     [(m ^^ zero = m)
880      (m ^^ succ n = m o (m ^^ n))]
881
882 let {m := (alist->fmap [[1 --> 2] [2 --> 3] [3 --> 1]]);
883     _ := (print "\nm iterated once: " (eval map->set m ^^ 1));
884     _ := (print "\nm iterated twice: " (eval map->set m ^^ 2));
885     _ := (print "\nm iterated thrice: " (eval map->set m ^^ 3))}
886 (print "\nAre m and m^^3 identical?: " (eval m = m ^^ 3))
887
888 declare compose2: (S) [(Map S S) (Map S S)] -> (Map S S) [[alist->fmap alist->fmap]]
889
890 assert* compose2-def :=
891     [(m compose2 empty-map = m)
892      (m compose2 [k v] ++ more = [k v'] ++ (m compose2 more) <== m applied-to v = SOME v')
893      (m compose2 [k v] ++ more = [k v] ++ (m compose2 more) <== m applied-to v = NONE)]
894
895 define comp2-is-comm := (forall m1 m2 . m1 compose2 m2 = m2 compose2 m1)
896
897 (falsify comp2-is-comm 10)
898
899 define comp2-is-assoc := (forall m1 m2 m3 . m1 compose2 (m2 compose2 m3) = (m1 compose2 m2) compose2 m3)
900
901  #(falsify comp2-is-assoc 80)

```

```

902
903 (define comp2-app-lemma
904   (forall m1 m2 k v . (m2 compose2 m1) applied-to k = SOME v <==>
905     ((exists v' . m1 applied-to k = SOME v' & m2 applied-to v' = SOME v) |
906      (m1 applied-to k = NONE & m2 applied-to k = SOME v)))
907
908 # (falsify comp2-app-lemma 10)
909
910 declare compatible: (S, T) [(Map S T) (Map S T)] -> Boolean [<-> [alist->fmap alist->fmap]]
911
912 assert* compatible-def :=
913   [(m1 <-> m2 <==> agree-on m1 m2 (dom m1) /\ (dom m2))]
914
915 pick-any m
916   (!chain<- [(m <-> m)
917     <== (agree-on m m (dom m) /\ (dom m)) [compatible-def]
918     <== (agree-on m m dom m) [Set.intersection-lemma-3]
919     <== (m |^ dom m = m |^ dom m) [agree-on-def]])
920
921 (eval [[1 --> 'a] [2 --> 'b]] <-> [[1 --> 'a] [3 --> 'c]])
922
923 (eval [[1 --> 'a] [2 --> 'b]] <-> [[1 --> 'a] [2 --> 'foo] [3 --> 'c]])
924
925 define compatible-theorem-1 := (forall m . m <-> m)
926
927 (falsify compatible-theorem-1 20)
928
929 define compatible-theorem-2 := (forall m1 m2 . m1 <-> m2 <==> m2 <-> m1)
930
931 #(running-time (lambda () (falsify compatible-theorem-2 10)) 0)
932 # with new evall: 4.22
933
934 define compatible-theorem-3 := (forall m1 m2 m3 . m1 <-> m2 & m2 <-> m3 ==> m1 <-> m3)
935
936 (falsify compatible-theorem-3 10)
937
938 }
939
940
941 EOF
942 load "c:\\np\\lib\\basic\\fmaps"

```