

lib/main/map.ath

```

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

```

```

68 overload ++ update
69 #set-precedence ++ 210
70
71
72
73 define [key key1 key2 k k' k1 k2] := [?key ?key1 ?key2 ?k ?k' ?k1 ?k2]
74 define [val val1 val2 v v' v1 v2 x x1 x2 y y1 y2] :=
75   [?val ?val1 ?val2 ?v ?v' ?v1 ?v2 ?x ?x1 ?x2 ?y ?y1 ?y2]
76 define [m m' m1 m2 m3 rest rest1] := [?m:(Map 'S1 'S2) ?m':(Map 'S1 'S2) ?m1:(Map 'S3 'S4)
77   ?m2:(Map 'S5 'S6) ?m3:(Map 'S7 'S8) ?rest:(Map 'S9 'S10) ?rest1:(Map 'S11 'S12)]
78 define [S S1 S2 S3] := [?S:(Set.Set 'S) ?S1:(Set.Set 'S1) ?S2:(Set.Set 'S2) ?S3:(Set.Set 'S3)]
79
80 define [L L1 L2 more more1] := [?L ?L1 ?L2 ?more ?more1]
81
82 declare apply: (S, T) [(Map S T) S] -> (Option T) [applied-to 110 [alist->map id]]
83
84 define at := applied-to
85
86 declare remove: (S, T) [(Map S T) S] -> (Map S T) [- 120 [alist->map id]]
87
88 left-assoc -
89
90 (define t1 (- ?x ?y))
91
92 define (removed-from key map) := (remove map key)
93
94 #assert* remove-axioms :=
95 # [( _ removed-from empty-map = empty-map)
96 # (key removed-from [key _] ++ rest = key removed-from rest)
97 # (key != x ==> x removed-from [key val] ++ rest = [key val] ++ (x removed-from rest))]
98
99 assert* remove-def :=
100 [([] - _ = empty-map)
101  ([key _] ++ rest - key = rest - key)
102  (key != x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]
103
104 (define t2 (- ?x null))
105 (define t3 (- ?x ?y))
106
107 #define (- map key) := (key removed-from map)
108
109
110
111 define M := [[1 --> 'a] [2 --> 'b] [1 --> 'c]]
112 define ide-map := [['a --> 1] ['b --> 2] ['c --> 3] ['a --> 99]]
113 define ide-map' := [['b --> 2] ['c --> 3] ['a --> 1] ['a --> 99]]
114 define ide-map'' := [['b --> 2] ['c --> 3] ['a --> 1] ['d --> 4] ['a --> 99]]
115
116 # (set-flag m1style-fundef "on")
117
118 assert* apply-axioms :=
119 [([] at _ = NONE)
120  ([key val] ++ _ at x = SOME val <== key = x)
121  ([key _] ++ rest at x = rest at x <== key != x)]
122
123 #define applied-to := apply
124 ## The following gives the result NONE:(Option 'T286327), but it should be NONE:(Option Int)
125 # (set-flag m1style-fundef "on")
126 # (apply' empty-map:(Map Int Int) 1) [FIXED]
127
128 conclude apply-lemma-1 :=
129 (forall key val rest x .
130   [key val] ++ rest at x = NONE ==> rest at x = NONE)
131 pick-any key val rest x
132 let {m := ([key val] ++ rest);
133     hyp := (m at x = NONE);
134     goal := (rest at x = NONE)}
135 assume hyp
136 (!two-cases
137   (!chain [(key = x)

```

```

138      ==> (m at x = SOME val) [apply-axioms]
139      ==> (m at x != NONE) [option-results]
140      ==> (hyp & ~hyp) [augment]
141      ==> goal [prop-taut]]
142    (!chain [(key != x)
143      ==> (m at x = rest at x) [apply-axioms]
144      ==> (NONE = rest at x) [hyp]
145      ==> goal [sym]]))
146
147  conclude apply-lemma-2 :=
148    (forall k v rest x .
149      [k v] ++ rest at x != NONE <==> k = x | rest at x != NONE)
150  pick-any k v rest x
151    (!two-cases
152      assume case-1 := (k = x)
153      (!equiv assume hyp := ([k v] ++ rest at x != NONE)
154        (!chain-> [(k = x) ==> (k = x | rest at x != NONE) [alternate]])
155        assume (k = x | rest at x != NONE)
156          (!chain-> [(k v] ++ rest at x)
157            = ([x v] ++ rest at x) [(k = x)]
158            = (SOME v) [apply-axioms]
159            ==> ([k v] ++ rest at x != NONE) [option-results]))
160      assume case-2 := (k != x)
161      (!equiv assume hyp := ([k v] ++ rest at x != NONE)
162        (!chain-> [hyp
163          ==> (rest at x != NONE) [apply-axioms]
164          ==> (k = x | rest at x != NONE) [alternate]])
165        assume C := (k = x | rest at x != NONE)
166        (!cases C
167          assume (k = x)
168            (!from-complements ([k v] ++ rest at x != NONE) (k = x) (k != x))
169            (!chain [(rest at x != NONE) ==> ([k v] ++ rest at x != NONE) [apply-lemma-1]])))
170
171  conclude apply-lemma-3 :=
172    (forall m k v1 v2 . m at k = SOME v1 & m at k = SOME v2 ==> v1 = v2)
173  pick-any m k v1 v2
174    assume hyp := (m at k = SOME v1 & m at k = SOME v2)
175    (!chain-> [(SOME v1)
176      = (m at k)
177      = (SOME v2)
178      ==> (v1 = v2) [option-results]])
179
180  conclude remove-correctness :=
181    (forall m x . m - x at x = NONE)
182  by-induction remove-correctness {
183    (m as empty-map) =>
184      pick-any x
185        (!chain [([] - x at x)
186          = ([] at x) [remove-def]
187          = NONE [apply-axioms]])
188    | (m as (update (pair key val) rest)) =>
189      let {IH := (forall x . rest - x at x = NONE)}
190      pick-any x
191        (!two-cases
192          assume case1 := (key = x)
193          (!chain [(m - x at x)
194            = (m - key at key) [case1]
195            = (rest - x at x) [case1 remove-def]
196            = NONE [IH]])
197          assume case2 := (key != x)
198          (!chain [(m - x at x)
199            = ([key val] ++ (rest - x) at x) [remove-def]
200            = (rest - x at x) [apply-axioms]
201            = NONE [IH]]))
202    }
203
204  define (RC2-M goal p1 p2) :=
205    match [goal p1 p2] {
206      [ (~ (s = t)) (s = u) (~ (u = t)) ] =>
207        (!by-contradiction goal

```

```

208         assume (~ goal)
209         (!chain-> [(~ goal)
210           ==> (s = t) [dn]
211           ==> (u = t) [(s = u)]
212           ==> (u = t & u /= t) [augment]
213           ==> false [prop-taut]]))
214     }
215
216
217 conclude remove-correctness-2 :=
218   (forall m x y . x /= y ==> (m - x) at y = m at y)
219 by-induction remove-correctness-2 {
220   (m as empty-map) =>
221     pick-any x y
222     assume hyp := (x /= y)
223     (!chain [(m - x) at y
224       = (m at y) [remove-def]])
225 | (m as (update (pair key val) rest)) =>
226   let {IH := (forall x y . x /= y ==> (rest - x) at y = rest at y)}
227   pick-any x y
228   assume hyp := (x /= y)
229   (!two-cases
230     assume case1 := (key = x)
231     #let {lemma := (!CongruenceClosure.cc (key /= y) [case1 hyp])}
232     let {lemma := (!RC2-M (key /= y) case1 hyp)}
233     (!chain [(m - x) at y
234       = ((rest - x) at y) [(key = x) remove-def]
235       = (rest at y) [IH]
236       = (m at y) [apply-axioms]])
237     assume (key /= x)
238     (!two-cases
239       assume (key = y)
240       (!combine-equations
241         (!chain [(m - x) at y
242           = ([key val] ++ (rest - x)) at y [remove-def]
243           = (SOME val) [apply-axioms]])
244         (!chain [(m at y)
245           = (SOME val) [apply-axioms]])
246         assume (key /= y)
247         (!combine-equations
248           (!chain [(m - x) at y
249             = ([key val] ++ (rest - x)) at y [remove-def]
250             = ((rest - x) at y) [apply-axioms]
251             = (rest at y) [IH]])
252           (!chain [(m at y)
253             = (rest at y) [apply-axioms]])))
254   }
255
256 declare map->set: (S, T) [(Map S T)] -> (Set.Set (Pair S T)) [[alist->set]]
257
258 assert* map->set-def :=
259   [(map->set empty-map = null)
260     (map->set [k v] ++ rest = (k @ v) ++ map->set rest - k)]
261
262 assert* map-identity := (m1 = m2 <==> map->set m1 = map->set m2)
263
264 (eval map->set ide-map)
265 (eval (alist->set ide-map) = (alist->set ide-map'))
266 (eval (alist->set ide-map) = (alist->set ide-map''))
267
268 conclude opair-lemma :=
269   (forall x1 x2 y1 y2 A . x1 /= x2 ==> x1 @ y1 in A <==> x1 @ y1 in x2 @ y2 ++ A)
270 pick-any x1:'S x2:'S y1:'T y2:'T A:(Set.Set (Pair 'S 'T))
271   assume (x1 /= x2)
272   (!equiv (!chain [(x1 @ y1 in A)
273     ==> (x1 @ y1 in x2 @ y2 ++ A) [Set.in-lemma-3]])
274     (!chain [(x1 @ y1 in x2 @ y2 ++ A)
275       ==> (x1 @ y1 = x2 @ y2 | x1 @ y1 in A) [Set.in-def]
276       ==> ((x1 = x2 & y1 = y2) | x1 @ y1 in A) [(datatype-axioms "Pair")]
277       ==> (x1 = x2 | x1 @ y1 in A) [prop-taut])

```

```

278      ==> (x1 != x2 & (x1 = x2 | x1 @ y1 in A)) [augment]
279      ==> ((x1 != x2 & x1 = x2) | (x1 != x2 & x1 @ y1 in A)) [prop-taut]
280      ==> (false | (x1 != x2 & x1 @ y1 in A)) [prop-taut]
281      ==> (x1 != x2 & x1 @ y1 in A) [prop-taut]
282      ==> (x1 @ y1 in A) [right-and]]))
283
284
285
286 define ms-lemma-1a :=
287 pick-any x key val rest v
288   assume hyp := (x != key)
289   (!chain [(key _) ++ rest at x = SOME v)
290     <==> (rest at x = SOME v) [apply-axioms]])
291
292 #define ms-lemma-1b :=
293 # (forall m k v x . k @ v in map->set m & x != k ==> k @ v in map->set (m - x))
294
295 declare dom: (S, T) [(Map S T)] -> (Set.Set S) [[alist->map]]
296
297 assert* dom-axioms :=
298   [(dom empty-map = null)
299     (dom [k _] ++ rest = k ++ dom rest)]
300
301 transform-output eval [Set.set->alist map->alist]
302
303 (eval dom ide-map)
304
305 conclude dom-lemma-1 :=
306   (forall k v rest . k in dom [k v] ++ rest)
307 pick-any k v rest
308   (!chain-> [true ==> (k in k ++ dom rest) [Set.in-lemma-1]
309     ==> (k in dom [k v] ++ rest) [dom-axioms]])
310
311 conclude dom-lemma-2 :=
312   (forall m k v . dom m subset dom [k v] ++ m)
313 pick-any m k v
314   (!Set.subset-intro
315     pick-any x
316       (!chain [(x in dom m)
317         ==> (x in k ++ dom m) [Set.in-lemma-3]
318         ==> (x in dom [k v] ++ m) [dom-axioms]]))
319
320
321 conclude dom-characterization :=
322   (forall m k . k in dom m <==> m at k != NONE)
323 by-induction dom-characterization {
324   (m as empty-map) =>
325     pick-any k
326       (!equiv
327         (!chain [(k in dom m)
328           ==> (k in null) [dom-axioms]
329           ==> false [Set.NC]
330           ==> (m at k != NONE) [prop-taut]])
331         assume hyp := (m at k != NONE)
332         (!chain-> [true
333           ==> (m at k = NONE) [apply-axioms]
334           ==> (m at k = NONE & hyp) [augment]
335           ==> false [prop-taut]
336           ==> (k in dom m) [prop-taut]]))
337   | (m as (update (pair x y) rest)) =>
338     let {IH := (forall k . k in dom rest <==> rest at k != NONE)}
339     pick-any k
340       (!chain [(k in dom m)
341         <==> (k in x ++ dom rest) [dom-axioms]
342         <==> (k = x | k in dom rest) [Set.in-def]
343         <==> (k = x | rest at k != NONE) [IH]
344         <==> (x = k | rest at k != NONE) [sym]
345         <==> (m at k != NONE) [apply-lemma-2]])
346   }
347

```

```

348 conclude dom-lemma-3 := (forall m k . dom (m - k) subset dom m)
349 by-induction dom-lemma-3 {
350   (m as empty-map: (Map 'K 'V)) =>
351     pick-any k: 'K
352     (!Set.subset-intro
353       pick-any x: 'K
354       (!chain [(x in dom m - k)
355         ==> (x in dom empty-map) [remove-def]
356         ==> (x in null) [dom-axioms]
357         ==> false [Set.NC]
358         ==> (x in dom m) [prop-taut]]))
359 | (m as (update (pair key: 'K val: 'V) rest)) =>
360   pick-any k: 'K
361   let {IH := (!claim (forall k . dom rest - k subset dom rest));
362     IH1 := (!chain-> [true ==> (dom rest - key subset dom rest) [IH]]);
363     IH2 := (!chain-> [true ==> (dom rest - k subset dom rest) [IH]]);
364     (!Set.subset-intro
365       pick-any x: 'K
366       (!two-cases
367         assume (key = k)
368         (!chain [(x in dom m - k)
369           ==> (x in dom m - key) [(key = k)]
370           ==> (x in dom rest - key) [remove-def]
371           ==> (x in dom rest) [IH1 Set.SC]
372           ==> (x in key ++ dom rest) [Set.in-lemma-3]
373           ==> (x in dom m) [dom-axioms]])
374         assume case-2 := (key != k)
375         (!chain [(x in dom m - k)
376           ==> (x in dom [key val] ++ (rest - k)) [remove-def]
377           ==> (x in key ++ dom rest - k) [dom-axioms]
378           ==> (x = key | x in dom rest - k) [Set.in-def]
379           ==> (x = key | x in dom rest) [Set.SC IH2]
380           ==> (x in key ++ dom rest) [Set.in-def]
381           ==> (x in dom m) [dom-axioms]]))
382   }
383
384
385 declare size: (S, T) [(Map S T)] -> N [[alist->map]]
386
387 assert* size-axioms := [(size m = card dom m)]
388
389 transform-output eval [nat->int]
390
391 (eval size ide-map)
392
393 conclude ms-rec-lemma :=
394   (forall m k v . size (m - k) < size [k v] ++ m)
395
396 conclude ms-rec-lemma
397   pick-any m: (Map 'K 'V) key: 'K val: 'V
398   let {L1 := (!by-contradiction (~ key in dom m - key)
399     assume h := (key in dom m - key)
400     (!absurd (!chain-> [true ==> ((m - key) at key = NONE) [remove-correctness]]
401       (!chain-> [h ==> ((m - key) at key != NONE) [dom-characterization]]));
402     L2 := (!chain-> [true ==> (key in dom [key val] ++ m) [dom-lemma-1]]);
403     L3 := (!both (!chain-> [true ==> (dom m - key subset dom m) [dom-lemma-3]]
404       (!chain-> [true ==> (dom m subset dom [key val] ++ m) [dom-lemma-2]]));
405     L4 := (!chain-> [L3 ==> (dom m - key subset dom [key val] ++ m) [Set.subset-transitivity]]);
406     (!chain-> [L4 ==> (L4 & L2 & L1) [augment]
407       ==> (dom m - key proper-subset dom [key val] ++ m) [Set.proper-subset-lemma]
408       ==> (card dom m - key < card dom [key val] ++ m) [Set.proper-subset-card-theorem]
409       ==> (size m - key < size [key val] ++ m) [size-axioms]])
410   }
411 define ms-theorem :=
412   (forall m k v . k @ v in map->set m <==> m at k = SOME v)
413
414 (define (property m)
415   (forall k v . k @ v in map->set m <==> m at k = SOME v))
416
417 conclude ms-theorem

```

```

418 (!strong-induction.measure-induction ms-theorem size
419   pick-any m:(Map 'K 'V)
420   assume IH := (forall m' . size m' < size m ==> property m')
421   conclude (property m)
422   datatype-cases (property m) on m {
423     (em as empty-map:(Map 'K 'V)) =>
424       (pick-any k:'K v:'V
425         let {none := NONE:(Option 'V)}
426           (!equiv (!chain [(k @ v in map->set em)
427                         ==> (k @ v in null)
428                         ==> false
429                         ==> (em at k = SOME v)]))
430         assume hyp := (em at k = SOME v)
431           (!chain-> [true
432                     ==> (em at k = none) [apply-axioms]
433                     ==> (em at k = none & hyp) [augment]
434                     ==> (em at k = none & em at k != none) [option-results]
435                     ==> false [prop-taut]
436                     ==> (k @ v in map->set em) [prop-taut]]]))
437   | (map as (update (pair key:'K val:'V) rest)) =>
438     pick-any k:'K v:'V
439     let {goal := (k @ v in map->set map <==> map at k = SOME v);
440         lemma := (!chain-> [true ==> (size rest - key < size map) [ms-rec-lemma]
441                             ==> (size rest - key < size m) [(m = map)]]]}
442     (!two-cases
443       assume case1 := (k = key)
444       (!equiv assume hyp := (k @ v in map->set map)
445         let {D := (!chain-> [hyp
446                             ==> (k @ v in key @ val ++ map->set rest - key)
447                             ==> (key @ v in key @ val ++ map->set rest - key) [case1]
448                             ==> (key @ v = key @ val | key @ v in map->set rest - key) [Set.in]
449                             (!cases D
450                               assume h1 := (key @ v in map->set rest - key)
451                               let {_ := (!absurd (!chain-> [h1 ==> ((rest - key) at key = SOME v)
452                                                         ==> (NONE = SOME v)
453                                                         (!chain-> [true ==> (NONE != SOME v) [option-results]]))
454                               (!from-false (map at k = SOME v))
455                               assume h2 := (key @ v = key @ val)
456                               let {v=val := (!chain [h2 ==> (v = val)]]}
457                               (!chain-> [(map at key) = (SOME val) [apply-axioms]
458                                         = (SOME v) [v=val]
459                                         ==> (map at key = SOME v)
460                                         ==> (map at k = SOME v) [(k = key)]]])
461                               assume hyp := (map at k = SOME v)
462                               let {val=v := (!chain-> [(SOME val)
463                                                         = (map at key) [apply-axioms]
464                                                         = (map at k) [case1]
465                                                         = (SOME v) [hyp]
466                                                         ==> (val = v) [option-results]]]}
467                               (!chain-> [true ==> (key @ val in key @ val ++ map->set (rest - key)) [Set.in-lem]
468                                         ==> (key @ val in map->set map)
469                                         ==> (k @ val in map->set map)
470                                         ==> (k @ v in map->set map)
471                                         [val=v]]])
472                               assume case2 := (k != key)
473                               (!iff-comm
474                                 (!chain [(map at k = SOME v)
475                                           <==> (rest at k = SOME v) [apply-axioms]
476                                           <==> ((rest - key) at k = SOME v) [remove-correctness-2]
477                                           <==> (k @ v in map->set rest - key) [IH]
478                                           <==> (k @ v in key @ val ++ map->set rest - key) [(k @ v in map->set rest - key <==>
479                                                                 k @ v in key @ val ++ map->set rest
480                                                                 <== case2 [opair-lemma])
481                                           <==> (k @ v in map->set map) [map->set-def]]]))
482                               })
483     })

```

```

482
483 (eval dom ide-map)
484
485 conclude dom-characterization-2 :=
486   (forall m x . x in dom m <==> exists v . x @ v in map->set m)
487 pick-any m: (Map 'K 'V) x:'K
488   (!chain [(x in dom m)
489     <==> (m at x != NONE) [dom-characterization]
490     <==> (exists v . m at x = SOME v) [option-results]
491     <==> (exists v . x @ v in map->set m) [ms-theorem]])
492
493 conclude ms-corollary :=
494   (forall m k . m at k = NONE <==> ~ exists v . k @ v in map->set m)
495 pick-any m: (Map 'K 'V) k:'K
496   (!equiv (!chain [(m at k = NONE)
497     ==> (~ exists v . m at k = SOME v) [option-results]
498     ==> (~ exists v . k @ v in map->set m) [ms-theorem]])
499     (!chain [(~ exists v . k @ v in map->set m)
500       ==> (~ exists v . m at k = SOME v) [ms-theorem]
501       ==> (m at k = NONE) [option-results]]))
502
503
504 conclude identity-characterization-1 :=
505   (forall m1 m2 . m1 = m2 ==> forall k . m1 at k = m2 at k)
506 pick-any m1: (Map 'S 'T) m2: (Map 'S 'T)
507 assume hyp := (m1 = m2)
508 let {m1=m2 := (!chain-> [hyp ==> (map->set m1 = map->set m2) [map-identity]])}
509 pick-any k:'S
510   (!cases (!chain-> [true ==> (m1 at k = NONE | exists v . m1 at k = SOME v) [option-results]])
511     assume case1 := (m1 at k = NONE)
512     let {p := (!by-contradiction (m2 at k = NONE)
513       assume h := (m2 at k != NONE)
514       pick-witness v for (!chain-> [h ==> (exists v . m2 at k = SOME v) [option-results]]) wp
515       (!chain-> [wp ==> (k @ v in map->set m2) [ms-theorem]
516         ==> (k @ v in map->set m1) [m1=m2]
517         ==> (m1 at k = SOME v) [ms-theorem]
518         ==> (m1 at k != NONE) [option-results]
519         ==> (case1 & m1 at k != NONE) [augment]
520         ==> false [prop-taut]]))}
521     (!combine-equations (m1 at k = NONE) (m2 at k = NONE))
522     assume case2 := (exists v . m1 at k = SOME v)
523     pick-witness v for case2
524     (!combine-equations
525       (m1 at k = SOME v)
526       (!chain-> [(m1 at k = SOME v)
527         ==> (k @ v in map->set m1) [ms-theorem]
528         ==> (k @ v in map->set m2) [m1=m2]
529         ==> (m2 at k = SOME v) [ms-theorem]]))
530
531 conclude identity-characterization-2 :=
532   (forall m1 m2 . (forall k . m1 at k = m2 at k) ==> m1 = m2)
533 pick-any m1: (Map 'S 'T) m2: (Map 'S 'T)
534 assume hyp := (forall k . m1 at k = m2 at k)
535 let {m1=m2-as-sets :=
536   (!Set.set-identity-intro-direct
537     (!pair-converter
538       pick-any k:'S v:'T
539       (!chain [(k @ v in map->set m1)
540         <==> (m1 at k = SOME v) [ms-theorem]
541         <==> (m2 at k = SOME v) [hyp]
542         <==> (k @ v in map->set m2) [ms-theorem]]))}
543   (!chain-> [m1=m2-as-sets ==> (m1 = m2) [map-identity]])
544
545 conclude identity-characterization :=
546   (forall m1 m2 . m1 = m2 <==> forall k . m1 at k = m2 at k)
547 pick-any m1: (Map 'S 'T) m2: (Map 'S 'T)
548   (!equiv
549     (!chain [(m1 = m2) ==> (forall k . m1 at k = m2 at k) [identity-characterization-1]])
550     (!chain [(forall k . m1 at k = m2 at k) ==> (m1 = m2) [identity-characterization-2]]))
551

```



```

552
553 declare restricted-to: (S, T) [(Map S T) (Set.Set S)] -> (Map S T) [150 |^ [alist->map Set.alist->set]]
554
555 assert* restrict-axioms :=
556   [(empty-map |^ _ = empty-map)
557    (k in A ==> [k v] ++ rest |^ A = [k v] ++ (rest |^ A))
558    (~ k in A ==> [k v] ++ rest |^ A = rest |^ A)]
559
560 (eval [[1 --> 'a] [2 --> 'b] [3 --> 'c]] |^ [1 3])
561
562 conclude restriction-theorem-1 := (forall m A . dom m |^ A subset A)
563 by-induction restriction-theorem-1 {
564   empty-map =>
565     pick-any A
566     (!Set.subset-intro
567       pick-any x
568       (!chain [(x in dom empty-map |^ A)
569                ==> (x in dom empty-map) [restrict-axioms]
570                ==> (x in null) [dom-axioms]
571                ==> false [Set.NC]
572                ==> (x in A) [prop-taut]]))
573 | (m as (update (pair k v) rest)) =>
574   pick-any A
575   let {IH := (forall A . dom rest |^ A subset A);
576        lemma := (!chain-> [true ==> (dom rest |^ A subset A) [IH]])}
577   (!two-cases
578     assume case-1 := (k in A)
579     (!Set.subset-intro
580       pick-any x
581       (!chain [(x in dom m |^ A)
582                ==> (x in dom [k v] ++ (rest |^ A)) [restrict-axioms]
583                ==> (x in k ++ dom rest |^ A) [dom-axioms]
584                ==> (x = k | x in dom rest |^ A) [Set.in-def]
585                ==> (x in A | x in dom rest |^ A) [case-1]
586                ==> (x in A | x in A) [Set.SC]
587                ==> (x in A) [prop-taut]]))
588     assume case-2 := (~ k in A)
589     (!Set.subset-intro
590       pick-any x
591       (!chain [(x in dom m |^ A)
592                ==> (x in dom rest |^ A) [restrict-axioms]
593                ==> (x in A) [Set.SC]]))
594 }
595
596
597 conclude restriction-theorem-2 :=
598   (forall m A . dom m subset A ==> m |^ A = m)
599 by-induction restriction-theorem-2 {
600   (m as empty-map) =>
601     pick-any A
602     assume hyp := (dom m subset A)
603     (!chain [(m |^ A) = m [restrict-axioms]])
604 | (m as (update (pair key val) rest)) =>
605   pick-any A
606   assume hyp := (dom m subset A)
607   let {lemma1 := (!chain-> [true ==> (key in dom m) [dom-lemma-1]
608                             ==> (key in A) [Set.SC]]);
609        lemma2 := (!chain-> [true ==> (dom rest subset dom m) [dom-lemma-2]
610                             ==> (dom rest subset dom m & hyp) [augment]
611                             ==> (dom rest subset A) [Set.subset-transitivity])};
612   IH := (forall A . dom rest subset A ==> rest |^ A = rest)
613   (!chain [(m |^ A)
614            = ([key val] ++ (rest |^ A)) [restrict-axioms]
615            = ([key val] ++ rest) [IH]])
616 }
617
618
619 declare range: (S, T) [(Map S T)] -> (Set.Set T) [[alist->map]]
620
621 assert* range-def :=

```

```

622   [(range m = Set.range map->set m)]
623
624 (eval range ide-map)
625
626 conclude range-lemma-1 :=
627   (forall m v . v in range m <==> exists k . k @ v in map->set m)
628 pick-any m v
629   (!chain [(v in range m)
630     <==> (v in Set.range map->set m) [range-def]
631     <==> (exists k . k @ v in map->set m) [Set.range-characterization]])
632
633 conclude range-characterization :=
634   (forall m v . v in range m <==> exists k . m at k = SOME v)
635 pick-any m v
636   (!chain [(v in range m)
637     <==> (exists k . k @ v in map->set m) [range-lemma-1]
638     <==> (exists k . m at k = SOME v) [ms-theorem]])
639
640 conclude range-lemma-2 :=
641   (forall k v rest . v in range [k v] ++ rest)
642 pick-any k v rest
643   (!chain<- [(v in range [k v] ++ rest)
644     <== (v in Set.range map->set [k v] ++ rest) [range-def]
645     <== (v in Set.range k @ v ++ map->set rest - k) [map->set-def]
646     <== (v in v ++ Set.range map->set rest - k) [Set.range-def]
647     <== (v = v | v in Set.range map->set rest - k) [Set.in-def]
648     <== (v = v) [alternate]])
649
650 define range-lemma-conjecture :=
651   (forall m k v . range m subset range [k v] ++ m)
652
653  #(falsify range-lemma-conjecture 10)
654
655 conclude removal-range-theorem :=
656   (forall m k . range m - k subset range m)
657 pick-any m k
658   (!Set.subset-intro
659     pick-any v
660     assume hyp := (in v range m - k)
661     pick-witness key for
662       (!chain<- [(exists key . m - k at key = SOME v)
663         <== hyp [range-characterization]])
664       key-premise
665       let {k!=key} :=
666         (!by-contradiction (k != key)
667           assume (k = key)
668             (!absurd (!chain-> [key-premise
669               ==> (m - key at key = SOME v) [(k = key)]])
670               (!chain-> [true ==> (m - key at key = NONE) [remove-correctness]
671                 ==> (m - key at key != SOME v) [option-results]])))
672             (!chain-> [k!=key ==> (m - k at key = m at key) [remove-correctness-2]
673               ==> (SOME v = m at key) [key-premise]
674               ==> (m at key = SOME v) [sym]
675               ==> (exists key . m at key = SOME v) [existence]
676               ==> (v in range m) [range-characterization]]))
677
678 declare range-restricted: (S, T) [(Map S T) (Set.Set T)] -> (Map S T) [150 ^| [alist->map Set.alist->set]]
679
680 assert* range-restricted-def :=
681   [(empty-map ^| _ = empty-map)
682     ([k v] ++ rest ^| A = [k v] ++ (rest - k ^| A) <== v in A)
683     ([k v] ++ rest ^| A = rest - k ^| A <== ~ v in A)]
684
685 (define p (forall m A . range m ^| A subset range m))
686 define eye-color := [['bob --> 'brown] ['tom --> 'blue] ['lisa --> 'green]
687   ['peter --> 'blue] ['ann --> 'brown]]
688
689 (eval eye-color ^| ['blue])
690
691 (define vpf

```

```

692 (method (goal premises)
693   (!vprove-from goal premises [['poly true] ['subsorting false] ['max-time 3000]])))
694
695 (define spf
696   (method (goal premises)
697     (!sprove-from goal premises [['poly true] ['subsorting false] ['max-time 300]])))
698
699 ### CAUTION: THE PATTERN (m as null) seemed to work!
700
701 (define range-restriction-theorem-1 :=
702   (forall m A . range m ^| A subset range m)
703
704 (declare agree-on: (S, T) [(Map S T) (Map S T) (Set.Set S)] -> Boolean
705   [[alist->map alist->map Set.alist->set]]
706
707 (assert* agree-on-def := [(agree-on m1 m2 A) <==> m1 |^ A = m2 |^ A])
708
709 (eval (agree-on ide-map ide-map ['a 'b]))
710
711 (eval (agree-on [['a --> 1] ['b --> 2]]
712   [['b --> 3] ['a --> 1]]
713   ['b]))
714
715 (declare override: (S, T) [(Map S T) (Map S T)] -> (Map S T) [** [alist->map alist->map]])
716
717 (assert* override-def :=
718   [(m ** [] = m)
719    (m ** [k v] ++ rest = [k v] ++ (m ** rest))]
720
721 (eval [[1 --> 'a] [2 --> 'b]] ** [[1 --> 'foo] [3 --> 'c]])
722
723
724 (conclude override-theorem-1 := (forall m . [] ** m = m)
725 (by-induction override-theorem-1 {
726   (m as empty-map) =>
727     (!chain [(empty-map ** m) = empty-map [override-def]])
728 | (m as (update (pair k v) rest)) =>
729   let {IH := ([ ] ** rest = rest)}
730     (!chain [(empty-map ** m)
731               = ([k v] ++ (empty-map ** rest)) [override-def]
732               = ([k v] ++ rest) [IH]])
733 }
734
735 (define conj1 := (forall m1 m2 . dom m2 ** m1 = (dom m2) \ / (dom m1))
736
737 (by-induction (forall m1 m2 . dom m2 ** m1 = (dom m2) \ / (dom m1)) {
738   (m1 as empty-map: (Map 'K 'V)) =>
739     pick-any m2: (Map 'K 'V)
740       (!chain [(dom m2 ** m1)
741                 = (dom m2) [override-def]
742                 = (null \ / dom m2) [Set.union-def]
743                 = ((dom m2) \ / null) [Set.union-commutes]
744                 = ((dom m2) \ / (dom m1)) [dom-axioms]])
745 | (m1 as (update (pair k: 'K v: 'V) rest)) =>
746   let {IH := (forall m2 . dom m2 ** rest = (dom m2) \ / (dom rest))}
747     pick-any m2: (Map 'K 'V)
748       (!chain [(dom m2 ** m1)
749                 = (dom [k v] ++ (m2 ** rest)) [override-def]
750                 = (k ++ dom (m2 ** rest)) [dom-axioms]
751                 = (k ++ ((dom m2) \ / (dom rest))) [IH]
752                 = ((dom m2) \ / k ++ dom rest) [Set.union-lemma-2]
753                 = ((dom m2) \ / dom m1) [dom-axioms]])
754 }
755
756 (define conj2 :=
757   (forall m1 m2 k . k in dom m1 ==> (m2 ** m1) at k = m1 at k)
758
759
760 # (falsify conj2 20)
761

```

```

762 by-induction conj2 {
763   (m1 as empty-map: (Map 'S 'T)) =>
764   pick-any m2: (Map 'S 'T) k: 'S
765     (!chain [(k in dom m1)
766               ==> (k in null) [dom-axioms]
767               ==> false [Set.NC]
768               ==> ((m2 ** m1) at k = m1 at k) [prop-taut]])
769 | (m1 as (update (pair key val) rest)) =>
770   let {IH := (forall m2 k . k in dom rest ==> (m2 ** rest) at k = rest at k)}
771   pick-any m2 k
772     assume hyp := (k in dom m1)
773     (!cases (!chain-> [hyp
774                       ==> (k in key ++ dom rest) [dom-axioms]
775                       ==> (k = key | k in dom rest) [Set.in-def]
776                       ==> (k = key | k /= key & k in dom rest) [prop-taut]])
777       assume (k = key)
778       (!chain [(m2 ** m1) at k)
779               = ([key val] ++ (m2 ** rest) at k) [override-def]
780               = ([key val] ++ (m2 ** rest) at key) [(k = key)]
781               = (SOME val) [apply-axioms]
782               = (m1 at key) [apply-axioms]
783               = (m1 at k) [(k = key)]])
784       assume (k /= key & k in dom rest)
785       (!chain [(m2 ** m1) at k)
786               = ([key val] ++ (m2 ** rest)) at k [override-def]
787               = ((m2 ** rest) at k) [apply-axioms]
788               = (rest at k) [IH]
789               = (m1 at k) [apply-axioms]]))
790 }
791
792 define conj3 := (forall m1 m2 . range m2 ** m1 = (range m2) \ / (range m1))
793 #(falsify conj3 10)
794 #(falsify conj3 20)
795
796 conclude restrict-theorem-3 :=
797   (forall m2 m1 A . (m1 ** m2) |^ A = m1 |^ A ** m2 |^ A)
798 by-induction restrict-theorem-3 {
799   (m2 as empty-map) =>
800   pick-any m1 A
801     (!combine-equations
802       (!chain [(m1 ** m2) |^ A = (m1 |^ A)])
803       (!chain [(m1 |^ A ** m2 |^ A)
804                 = (m1 |^ A ** empty-map)
805                 = (m1 |^ A)]))
806 | (m2 as (update (pair k v) rest)) =>
807   let {IH := (forall m1 A . (m1 ** rest) |^ A = m1 |^ A ** rest |^ A)}
808   pick-any m1 A
809     (!two-cases
810       assume (k in A)
811       (!combine-equations
812         (!chain [(m1 ** m2) |^ A)
813                 = ([k v] ++ (m1 ** rest)) |^ A [override-def]
814                 = ([k v] ++ ((m1 ** rest) |^ A)) [restrict-axioms]
815                 = ([k v] ++ (m1 |^ A ** rest |^ A)) [IH]])
816         (!chain [(m1 |^ A ** m2 |^ A)
817                 = (m1 |^ A ** [k v] ++ (rest |^ A)) [restrict-axioms]
818                 = ([k v] ++ (m1 |^ A ** rest |^ A)) [override-def]]))
819       assume (~ k in A)
820       (!chain [(m1 ** m2) |^ A)
821               = ([k v] ++ (m1 ** rest)) |^ A [override-def]
822               = ((m1 ** rest) |^ A) [restrict-axioms]
823               = (m1 |^ A ** rest |^ A) [IH]
824               = (m1 |^ A ** m2 |^ A) [restrict-axioms]]))
825 }
826
827 declare compose: (S1, S2, S3) [(Map S2 S3) (Map S1 S2)] -> (Map S1 S3) [o [alist->map alist->map]]
828
829 assert* compose-def :=
830   [(_ o empty-map = empty-map)
831    (m o [k v] ++ more = [k v'] ++ (m o more) <== m at v = SOME v')]

```

```

832     (m o [k v] ++ more = m o more <== m at v = NONE)]
833
834 define composition-is-comm := (forall m1 m2 . m1 o m2 = m2 o m1)
835
836 define composition-is-assoc := (forall m1 m2 m3 . m1 o (m2 o m3) = (m1 o m2) o m3)
837 define [n] := [?n:N]
838
839 declare iterate: (S, S) [(Map S S) N] -> (Map S S) [^^ [alist->map int->nat]]
840 define [^^ iterated] := [iterate iterate]
841
842 assert* iterate-axioms :=
843     [(m ^^ zero = m)
844      (m ^^ succ n = m o (m ^^ n))]
845
846 declare compose2: (S) [(Map S S) (Map S S)] -> (Map S S) [[alist->map alist->map]]
847
848 assert* compose2-def :=
849     [(m compose2 empty-map = m)
850      (m compose2 [k v] ++ more = [k v'] ++ (m compose2 more) <== m at v = SOME v')
851      (m compose2 [k v] ++ more = [k v] ++ (m compose2 more) <== m at v = NONE)]
852
853 define comp2-is-comm := (forall m1 m2 . m1 compose2 m2 = m2 compose2 m1)
854
855 define comp2-is-assoc := (forall m1 m2 m3 . m1 compose2 (m2 compose2 m3) = (m1 compose2 m2) compose2 m3)
856
857 (define comp2-app-lemma
858     (forall m1 m2 k v . (m2 compose2 m1) at k = SOME v <==>
859         ((exists v' . m1 at k = SOME v' & m2 at v' = SOME v) |
860          (m1 at k = NONE & m2 at k = SOME v))))
861
862 declare compatible: (S, T) [(Map S T) (Map S T)] -> Boolean [<-> [alist->map alist->map]]
863
864 assert* compatible-def :=
865     [(m1 <-> m2 <==> agree-on m1 m2 (dom m1) /\ (dom m2))]
866
867 pick-any m
868     (!chain<- [(m <-> m)
869                <== (agree-on m m (dom m) /\ (dom m)) [compatible-def]
870                <== (agree-on m m dom m) [Set.intersection-lemma-3]
871                <== (m | ^ dom m = m | ^ dom m) [agree-on-def]])
872
873 } # close module Map

```