lib/main/dmap.ath

```
# Module for rudimentary finite maps. This module is natively
# understood by the SMT translator, and it's how Athena handles
# SMT problems involving finite functions.

load "set"
load "strong-induction"
open Pair

module DMap {

define [null ++ in subset proper-subset \/ /\ \ card A B C] :=
       [Set.null Set.++ Set.in Set.subset Set.proper-subset
        Set.\/ Set./\ Set.\ Set.card
        ?A:(Set.Set 'S1) ?B:(Set.Set 'S2) ?C:(Set.Set 'S3)]

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

set-precedence empty-map 250

define (alist->dmap-general L preprocessor) :=
  match L {
    [d (some-list pairs)] =>
       letrec {loop := lambda (L)
                         match L {
                           [] => (empty-map d)
                         | (list-of (|| [x --> n] [x n]) rest) =>
                             (update (pair (preprocessor x) (preprocessor n)) (loop rest))}}
         (loop pairs)
  | _ => L
  }

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

define (dmap->alist-general m preprocessor) :=
  letrec {loop := lambda (m pairs)
                    match m {
                      (empty-map d) => [d (rev pairs)]
                    | (update (pair k v) rest) =>
                        (loop rest (add [(preprocessor k) --> (preprocessor v)] pairs))
                    | _ => m}}
    (loop m [])

(define (remove-from m k)
  (match m
    ((empty-map _) m)
    ((update (binding as (pair key val)) rest)
      (check ((equal? k key) (remove-from rest k))
             (else (update binding (remove-from rest k)))))))

define (dmap->alist-canonical-general m preprocessor) :=
  letrec {loop := lambda (m pairs)
                    match m {
                      (empty-map d) => [d (rev pairs)]
                    | (update (pair k v) rest) =>
                        (loop (remove-from rest k)
                              (add [(preprocessor k) --> (preprocessor v)] pairs))
                    | _ => m}}
    (loop m [])

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

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

declare apply: (K, V) [(DMap K V) K] -> V [110 [alist->dmap id]]

define [at] := [apply]

```

```
68 overload ++ update
69
70 set-precedence ++ 210
71
72 define [key k k' k1 k2 d d' val v v' v1 v2] := [?key ?k ?k' ?k1 ?k2 ?d ?d' ?val ?v ?v' ?v1 ?v2]
73 define [h t] := [Set.h Set.t]
74
75 define [m m' m1 m2 rest] := [?m:(DMap 'S1 'S2) ?m':(DMap 'S3 'S4) ?m1:(DMap 'S5 'S6) ?m2:(DMap 'S7 'S8) ?rest:(DMap
76
77 assert* apply-def :=
78   [(empty-map d at _ = d)
79    (k @ v ++ rest at x = v <== k = x)
80    (k @ v ++ rest at x =  rest at x <== k =/= x)]
81
82 ## Some testing:
83
84 define make-map :=
85   lambda (L)
86     match L {
87        [] => (empty-map 0)
88      | (list-of [x n] rest) => (update (x @ n) (make-map rest))
89     }
90
91 define update* :=
92   lambda (fm pairs)
93     letrec {loop := lambda (pairs res)
94                       match pairs {
95                         [] => res
96                       | (list-of [key val] more) => (loop more (update res key val))}}
97        (loop pairs fm)
98
99
100 declare default: (K, V) [(DMap K V)] -> V [200 [alist->dmap]]
101
102 assert* default-def :=
103   [(default empty-map d = d)
104    (default _ ++ rest = default rest)]
105
106 declare remove: (S, T) [(DMap S T) S] -> (DMap S T) [- 120 [alist->dmap id]]
107
108 left-assoc -
109
110 assert* remove-def :=
111   [(empty-map d - _ = empty-map d)
112    ([key _] ++ rest - key = rest - key)
113    (key =/= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]
114
115 declare dom: (S, T) [(DMap S T)] -> (Set.Set S)   [[alist->dmap]]
116
117 assert* dom-def :=
118   [(dom empty-map _ = null)
119    (dom [k v] ++ rest = dom rest - k <== v = default rest)
120    (dom [k v] ++ rest = k ++ dom rest <== v =/= default rest)]
121
122 declare size: (S, T) [(DMap S T)] -> N [[alist->dmap]]
123 assert* size-axioms := [(size m = card dom m)]
124
125 define rc1 := (forall m x . (m - x) at x = default m)
126
127 by-induction rc1 {
128   (m as (empty-map d)) =>
129     pick-any x
130      (!chain [(m - x at x)
131               = (m at x)       [remove-def]
132               = d              [apply-def]
133               = (default m)    [default-def]])
134 | (m as (update (pair k:'S v) rest)) =>
135    let {IH := (forall x . rest - x at x = default rest)}
136      pick-any x:'S
137        (!two-cases
```

```
138             assume (k = x)
139               (!chain [(m - x at x)
140                        = (m - k at k)    [(k = x)]
141                        = (rest - k at k)    [remove-def]
142                        = (default rest)     [IH]
143                        = (default m)        [default-def]
144                         ])
145             assume (k =/= x)
146               (!chain [(m - x at x)
147                        = ((k @ v) ++ (rest - x) at x)  [remove-def]
148                        = (rest - x at x)               [apply-def]
149                        = (default rest)                [IH]
150                        = (default m)                   [default-def]]))
151 }
152
153 define rc2 := (forall m k x . k =/= x ==> m - k at x = m at x)
154
155 by-induction rc2 {
156   (m as (empty-map d:'V)) =>
157     pick-any k:'K x:'K
158       assume (k =/= x)
159         let {L := (m - k at x);
160              R := (m at x)}
161           (!chain [L
162                    = (m at x)  [remove-def]])
163 | (m as (update (pair key:'K val:'V) rest:(DMap 'K 'V))) =>
164     pick-any k:'K x:'K
165       assume (k =/= x)
166         let {IH := (forall k x . k =/= x ==> (rest - k) at x = rest at x)}
167           (!two-cases
168             assume (key = k)
169               let {_ := (!by-contradiction (key =/= x)
170                            (!chain [(key = x)
171                                     ==> (k = x)            [(key = k)]
172                                     ==> (k = x & k =/= x)  [augment]
173                                     ==> false              [prop-taut]]))}
174               (!chain [(m - k at x)
175                        = (((k @ val) ++ rest) - k at x)  [(key = k)]
176                        = (rest - k at x)  [remove-def]
177                        = (rest at x)      [IH]
178                        = (m at x)         [apply-def]])
179             assume (key =/= k)
180               (!two-cases
181                  assume (x = key)
182                    (!chain [(m - k at x)
183                             = (([key val] ++ (rest - k)) at x)   [remove-def]
184                             = (([x val] ++ (rest - k)) at x)   [(x = key)]
185                             = val                              [apply-def]
186                             = (([x val] ++ rest) at x)         [apply-def]
187                             = (m at x)                          [(x = key)]])
188                  assume (x =/= key)
189                    (!chain [(m - k at x)
190                             = (([key val] ++ (rest - k)) at x)   [remove-def]
191                             = (rest - k at x)                    [apply-def]
192                             = (rest at x)                        [IH]
193                             = (m at x)                           [apply-def]])))
194 }
195
196 define rc3 := (forall m k . default m = default m - k)
197 by-induction rc3 {
198   (m as (empty-map d:'V)) =>
199     pick-any k
200       (!chain [(default m)
201                = (default m - k) [remove-def]])
202  | (m as (update (pair key:'K val:'V) rest)) =>
203     let {IH := (forall k . default rest = default rest - k)}
204     pick-any k:'K
205       (!two-cases
206          assume (key = k)
207           (!combine-equations
```

```
208               (!chain [(default m)
209                        = (default rest)       [default-def]
210                        = (default rest - k)   [IH]])
211               (!chain [(default m - k)
212                        = (default rest - k)   [remove-def]]))
213             assume (key =/= k)
214               (!chain-> [(default m - k)
215                          = (default key @ val ++ rest - k) [remove-def]
216                          = (default rest - k)              [default-def]
217                          = (default rest)                  [IH]
218                          = (default m)                     [default-def]
219                         ==> (default m - k = default m)
220                         ==> (default m = default m - k)     [sym]]))
221 }
222
223 conclude dom-lemma-1 :=
224   (forall k v rest . v =/= default rest ==> k in dom [k v] ++ rest)
225 pick-any k v rest
226   assume hyp := (v =/= default rest)
227   (!chain-> [true ==> (k in k ++ dom rest)     [Set.in-lemma-1]
228                   ==> (k in dom [k v] ++ rest) [dom-def]])
229
230 conclude dom-lemma-2 :=
231   (forall m k v . v =/= default m ==> dom m subset dom [k v] ++ m)
232 pick-any m k v
233   assume hyp := (v =/= default m)
234   (!Set.subset-intro
235      pick-any x
236        (!chain [(x in dom m)
237                 ==> (x in k ++ dom m)      [Set.in-lemma-3]
238                 ==> (x in dom [k v] ++ m)  [dom-def]]))
239
240 conclude dom-lemma-2b :=
241   (forall m x k v . v =/= default m & x in dom m ==> x in dom [k v] ++ m)
242 pick-any m x k v
243   assume (v =/= default m & x in dom m)
244   let {_ := (!chain-> [(v =/= default m) ==> (dom m subset dom [k v] ++ m) [dom-lemma-2]])}
245     (!chain-> [(x in dom m) ==> (x in dom [k v] ++ m) [Set.SC]])
246
247 # conclude dom-lemma-2c :=
248 #   (forall m x k v . x in dom [k v] ++ m ==> x = k | x in dom m - k)
249 # pick-any m:(DMap 'K 'V) x:'K k:'K v:'V
250 #   assume hyp := (x in dom [k v] ++ m)
251 #     (!two-cases
252 #       assume (v = default m)
253 #         (!chain-> [hyp
254 #                ==> (x in dom m - k)          [dom-def]
255 #                ==> (x = k | x in dom m - k) [prop-taut]])
256 #       assume (v =/= default m)
257 #         (!chain-> [hyp
258 #                ==> (x in k ++ dom m)          [dom-def]
259 #                ==> (x = k | x in dom m - k)       [Set.in-def]]))
260
261 define [< <=] := [N.< N.<=]
262 declare len: (S, T) [(DMap S T)] -> N [[alist->dmap]]
263
264 assert* len-def :=
265   [(len empty-map _ = zero)
266    (len _ @ _ ++ rest = S len rest)]
267
268 define len-lemma-1 :=
269   (forall m k v . len m < len (k @ v) ++ m)
270
271 by-induction len-lemma-1 {
272   (m as (empty-map d:'V)) =>
273     pick-any k v
274       let {len-left :=  (!chain [(len m) = zero                [len-def]]);
275            len-right := (!chain [(len k @ v ++ m) = (S len m) [len-def]])}
276         (!chain-> [true
277                ==> (zero < S len m)         [N.Less.<-def]
```

```
278               ==> (len m < len k @ v ++ m) [len-left len-right]])
279  | (m as (update (pair key:'K val:'V) rest)) =>
280      let {IH := (forall k v . len rest < len k @ v ++ rest)}
281        pick-any k:'K v:'V
282          let {len-left := (!chain [(len m)
283                                    = (S len rest)  [len-def]]);
284               len-right := (!chain [(len k @ v ++ m)
285                                     = (S len m)      [len-def]
286                                     = (S S len rest) [len-left]])}
287           (!chain-> [true
288                 ==> (S len rest < S S len rest)  [N.Less.<S]
289                 ==> (len m < len k @ v ++ m)     [len-left len-right]])
290  }
291
292  conclude len-lemma-2 := (forall m k . len m - k <= len m)
293  by-induction len-lemma-2 {
294    (m as (empty-map d:'V)) =>
295      pick-any k
296      (!chain-> [(len m - k)
297                 = (len m)             [remove-def]
298              ==>  (len m - k <= len m) [N.Less=.<=-def]])
299   | (m as (update (pair key:'K val:'V) rest)) =>
300      pick-any k:'K
301        let {IH := (forall k . len rest - k <= len rest);
302             L2 := (!chain-> [true ==> (len rest - k <= len rest) [IH]]);
303             L3 := (!chain-> [true ==> (len rest < len m)         [len-lemma-1]]);
304             L4 := (!chain-> [L2 ==> (L2 & L3)                    [augment]
305                              ==> (len rest - k < len m)          [N.Less=.transitive2]])}
306         (!two-cases
307           assume (key = k)
308             (!chain-> [(len m - k)
309                        = (len rest - k)                    [remove-def]
310                     ==> (len m - k <= len rest - k)        [N.Less=.<=-def]
311                     ==> (len m - k <= len rest - k & L2)   [augment]
312                     ==> (len m - k <= len rest)            [N.Less=.transitive]
313                     ==> (len m - k <= len rest & L3)       [augment]
314                     ==> (len m - k < len m)                [N.Less=.transitive2]
315                     ==> (len m - k <= len m)               [N.Less=.<=-def]])
316           assume (key =/= k)
317             let {L5 := (!chain-> [(len m - k)
318                                   = (len [key val] ++ (rest - k)) [remove-def]
319                                   = (S len rest - k)              [len-def]])}
320
321                (!chain-> [L4
322                       ==> (S len rest - k <= len m)  [N.Less=.discrete]
323                       ==> (len m - k <= len m)       [L5]]))
324  }
325
326  define len-lemma-3 :=
327     (forall key val k rest . len rest - k < len key @ val ++ rest)
328
329  conclude len-lemma-3
330    pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
331      let {m := (key @ val ++ rest);
332           L := (!chain-> [true
333                       ==> (len rest - k <= len rest) [len-lemma-2]])}
334        (!chain-> [true
335               ==> (len rest < len m)          [len-lemma-1]
336               ==> (L & len rest < len m)      [augment]
337               ==> (len rest - k < len m)      [N.Less=.transitive2]])
338
339  transform-output eval [nat->int]
340
341  define (lemma-D-property m) :=
342    (forall k .  k in dom m <==> m at k =/= default m)
343
344  define lemma-D := (forall m k . k in dom m <==> m at k =/= default m)
345
346  define lemma-D :=
347    (forall m . lemma-D-property m)
```

```
348
349 (!strong-induction.measure-induction lemma-D len
350 pick-any m:(DMap 'K 'V)
351   assume IH := (forall m' . len m' < len m ==> lemma-D-property m')
352     conclude (lemma-D-property m)
353       datatype-cases (lemma-D-property m) on m  {
354         (em as (empty-map d:'V)) =>
355          pick-any k
356            (!equiv
357              (!chain [(k in dom em)
358                   ==> (k in null)   [dom-def]
359                   ==> false         [Set.NC]
360                   ==> (em at k =/= default em) [prop-taut]])
361              assume h := (em at k =/= default em)
362                (!by-contradiction (k in dom em)
363                  assume (~ k in dom em)
364                    (!absurd (!reflex (default em))
365                             (!chain-> [h ==> (d =/= default em)          [apply-def]
366                                        ==> (default em =/= default em) [default-def]])))))
367       | (map as (update (pair key:'K val:'V) rest)) =>
368           pick-any k:'K
369             let {lemma1 := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
370                                        ==> (len rest - key < len m)   [(m = map)]]);
371                  lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
372                                        ==> (len rest < len m)   [(m = map)]])}
373              (!equiv
374                assume hyp := (k in dom map)
375                 (!two-cases
376                   assume (val = default rest)
377                     let {L1 := (!by-contradiction (k =/= key)
378                                   assume (k = key)
379                                    (!absurd
380                                     (!chain [(rest - key at key)
381                                            = (default rest)  [rc1]
382                                            = (default rest - key) [rc3]])
383                                     (!chain-> [(k in dom map)
384                                            ==> (key in dom map)        [(k = key)]
385                                            ==> (key in dom rest - key) [dom-def]
386                                            ==> (rest - key at key =/= default rest - key) [IH]])));
387                          _ := (!ineq-sym L1)}
388                       (!chain-> [(k in dom map)
389                              ==> (k in dom rest - key)  [dom-def]
390                              ==> (rest - key at k =/= default rest - key)  [IH]
391                              ==> (rest - key at k =/= default rest)        [rc3]
392                              ==> (rest - key at k =/= default map)         [default-def]
393                              ==> (rest at k =/= default map)               [rc2]
394                              ==> (map at k =/= default map)               [apply-def]])
395                   assume case2 := (val =/= default rest)
396                    let {M := method ()
397                               (!chain-> [(map at k) = (map at key)     [(k = key)]
398                                                     = val              [apply-def]
399                                     ==> (map at k =/= default rest)   [case2]
400                                     ==> (map at k =/= default map)    [default-def]])}
401                     (!cases (!chain-> [hyp
402                                     ==> (k in key ++ dom rest)    [dom-def]
403                                     ==> (k = key | k in dom rest) [Set.in-def]])
404                        assume (k = key)
405                          (!M)
406                        assume (k in dom rest)
407                          (!two-cases
408                            assume (k = key)
409                              (!M)
410                            assume (k =/= key)
411                              (!chain-> [(k in dom rest)
412                                     ==> (rest at k =/= default rest) [IH]
413                                     ==> (map at k  =/= default rest) [apply-def]
414                                     ==> (map at k  =/= default map)  [default-def]])))))
415                assume hyp := (map at k =/= default map)
416                  (!two-cases
417                     assume case1 := (val = default rest)
```

```
418                     let {k=/=key := (!by-contradiction (k =/= key)
419                                       assume (k = key)
420                                        let {p := (!chain [(map at k)
421                                                           = (map at key)    [(k = key)]
422                                                           = val             [apply-def]
423                                                           = (default rest)  [case1]
424                                                           = (default map)   [default-def]])}
425                                        (!absurd p hyp))}
426                       (!chain-> [hyp
427                              ==> (rest at k =/= default map) [apply-def]
428                              ==> ((rest - key) at k =/= default map) [rc2]
429                              ==> ((rest - key) at k =/= default rest) [default-def]
430                              ==> ((rest - key) at k =/= default rest - key) [rc3]
431                              ==> (k in dom rest - key)                      [IH]
432                              ==> (k in dom map)                             [dom-def]])
433                     assume case2 := (val =/= default rest)
434                         (!two-cases
435                           assume (k = key)
436                             (!chain<- [(k in dom map)
437                                    <== (key in dom map) [(k = key)]
438                                    <== (key in key ++ dom rest) [dom-def]
439                                    <== true                     [Set.in-lemma-1]])
440                           assume (k =/= key)
441                             (!chain-> [hyp
442                                    ==> (rest at k =/= default map)  [apply-def]
443                                    ==> (rest at k =/= default rest) [default-def]
444                                    ==> (k in dom rest)              [IH]
445                                    ==> (k = key | k in dom rest)    [prop-taut]
446                                    ==> (k in key ++ dom rest)       [Set.in-def]
447                                    ==> (k in dom map)               [dom-def]])))
448         )
449       })
450
451 conclude rc0 := (forall m x . ~ x in dom m - x)
452   pick-any m:(DMap 'K 'V)  x:'K
453     (!by-contradiction (~ x in dom m - x)
454       assume hyp := (x in dom m - x)
455        (!absurd (!chain-> [true ==> (m - x at x = default m) [rc1]])
456                 (!chain-> [hyp
457                        ==> (m - x at x =/= default m - x)   [lemma-D]
458                        ==> (m - x at x =/= default m)       [rc3]])))
459
460 conclude dom-lemma-3 := (forall m k . dom (m - k) subset dom m)
461 pick-any m:(DMap 'K 'V) k:'K
462 (!Set.subset-intro
463   pick-any x:'K
464     assume hyp := (x in dom m - k)
465       (!two-cases
466          assume (x = k)
467           let {L := (!chain-> [true ==> (m - k at k = default m) [rc1]])}
468              (!chain-> [hyp
469                     ==> (k in dom m - k)                [(x = k)]
470                     ==> (m - k at k =/= default m - k)  [lemma-D]
471                     ==> (m - k at k =/= default m)      [rc3]
472                     ==> (L & m - k at k =/= default m)  [augment]
473                     ==> false                           [prop-taut]
474                     ==> (x in dom m)                    [prop-taut]])
475          assume (x =/= k)
476            (!chain-> [hyp
477                   ==> (m - k at x =/= default m - k)   [lemma-D]
478                   ==> (m at x =/= default m - k)       [rc2]
479                   ==> (m at x =/= default m)           [rc3]
480                   ==> (x in dom m)                     [lemma-D]])))
481
482 conclude dom-corrolary-1 :=
483   (forall key val k rest  . k in dom rest - key ==> k in dom [key val] ++ rest)
484 pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
485    let {L1 := (!chain-> [true ==> (dom rest - key subset dom rest)         [dom-lemma-3]])}
486      (!two-cases
487        assume (val = default rest)
```

```
          (!chain [(k in dom rest - key)
               ==> (k in dom [key val] ++ rest) [dom-def]])
        assume (val =/= default rest)
          (!chain [(k in dom rest - key)
               ==> (k in dom rest)               [Set.SC]
               ==> (k = key | k in dom rest)     [prop-taut]
               ==> (k in key ++ dom rest)        [Set.in-def]
               ==> (k in dom [key val] ++ rest) [dom-def]]))

declare dmap->set: (K, V) [(DMap K V)] -> (Set.Set (Pair K V)) [[alist->dmap]]

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

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

(define (ms-lemma-1-property m)
  (forall k v . k @ v in dmap->set m ==> k in dom m))

(define ms-lemma-1
   (forall m (ms-lemma-1-property m)))

(!strong-induction.measure-induction ms-lemma-1 len
  pick-any m:(DMap 'K 'V)
    assume IH := (forall m' . len m' < len m ==> ms-lemma-1-property m')
      conclude (ms-lemma-1-property m)
           datatype-cases (ms-lemma-1-property m) on m  {
             (em as (empty-map d:'V)) =>
               pick-any k v:'V
                 (!chain [(k @ v in dmap->set em)
                      ==> (k @ v in Set.null)     [dmap->set-def]
                      ==> false                    [Set.NC]
                      ==> (k in dom em)            [prop-taut]])
         | (map as (update (pair key:'K val:'V) rest)) =>
              pick-any k:'K v:'V
               let {goal := (k @ v in dmap->set map ==> k in dom map);
                    lemma1 := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
                                        ==> (len rest - key < len m)   [(m = map)]]);
                    lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
                                        ==> (len rest < len m)   [(m = map)]])}
                 (!two-cases
                   assume C1 := (val = default rest)
                     (!chain [(k @ v in dmap->set map)
                          ==> (k @ v in dmap->set rest - key)  [dmap->set-def]
                          ==> (k in dom rest - key)            [IH]
                          ==> (k in dom map)                   [dom-def]])
                   assume C2 := (val =/= default rest)
                     let {_ := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]])}
                     (!chain [(k @ v in dmap->set map)
                          ==> (k @ v in key @ val ++ dmap->set rest - key) [dmap->set-def]
                          ==> (k @ v = key @ val | k @ v in dmap->set rest - key) [Set.in-def]
                          ==> (k = key & v = val | k @ v in dmap->set rest - key) [pair-axioms]
                          ==> (k = key | k @ v in dmap->set rest - key)           [prop-taut]
                          ==> (k = key | k in dom rest - key)                     [IH]
                          ==> (k = key | k in dom rest)                           [Set.SC]
                          ==> (k in key ++ dom rest)                              [Set.in-def]
                          ==> (k in dom map)                                      [dom-def]])
           )
         })

# conclude dom-corrolary-1 :=
#   (forall key val k rest  . k in dom rest - key ==> k in dom [key val] ++ rest)
# pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
#    let {L1 := (!chain-> [true ==> (dom rest - key subset dom rest)         [dom-lemma-3]])}
```

```
#       (!two-cases
#         assume (val = default rest)
#           (!chain [(k in dom rest - key)
#                ==> (k in dom [key val] ++ rest) [dom-def]])
#         assume (val =/= default rest)
#           (!chain [(k in dom rest - key)
#                ==> (k in dom rest)               [Set.SC]
#                ==> (k = key | k in dom rest)     [prop-taut]
#                ==> (k in key ++ dom rest)        [Set.in-def]
#                ==> (k in dom [key val] ++ rest) [dom-def]]))

assert* dmap-identity :=
 (forall m1 m2 . m1 = m2 <==> default m1 = default m2 & dmap->set m1 = dmap->set m2)

define dmap-identity-characterization :=
 (forall m1 m2 . m1 = m2 <==> forall k . m1 at k = m2 at k)

declare agree-on: (S, T) [(DMap S T) (DMap S T) (Set.Set S)] -> Boolean
                         [[alist->dmap alist->dmap Set.alist->set]]


assert* agree-on-def :=
  [(agree-on m1 m2 null)
   ((agree-on m1 m2 h Set.++ t) <==> m1 at h = m2 at h & (agree-on m1 m2 t))]

define agreement-characterization :=
  (forall A m1 m2 . (agree-on m1 m2 A) <==> forall k . k in A ==> m1 at k = m2 at k)

by-induction agreement-characterization {
  (A as Set.null:(Set.Set 'K)) =>
    pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
      let {p1 := assume (agree-on m1 m2 A)
                  pick-any k:'K
                    (!chain [(k in A)
                         ==> false                [Set.NC]
                         ==> (m1 at k = m2 at k)  [prop-taut]]);
           p2 := assume (forall k . k in A ==> m1 at k = m2 at k)
                  (!chain-> [true ==> (agree-on m1 m2 A) [agree-on-def]])}
       (!equiv p1 p2)
| (A as (Set.insert h:'K t:(Set.Set 'K))) =>
    let {IH := (forall m1 m2 . (agree-on m1 m2 t) <==> forall k . k in t ==> m1 at k = m2 at k)}
    pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
      let {p1 := assume hyp := (agree-on m1 m2 A)
                  pick-any k:'K
                    assume (k in A)
                      (!cases (!chain-> [(k in A)
                                         ==> (k = h | k in t)  [Set.in-def]])
                        assume (k = h)
                          (!chain-> [hyp
                                 ==> (m1 at h = m2 at h)    [agree-on-def]
                                 ==> (m1 at k = m2 at k)    [(k = h)]])
                        assume (k in t)
                         let {P := (!chain-> [hyp
                                              ==> (agree-on m1 m2 t)                          [agree-on-def]
                                              ==> (forall k . k in t ==> m1 at k = m2 at k) [IH]])}
                          (!chain-> [(k in t) ==> (m1 at k = m2 at k) [P]]));
           p2 := assume hyp := (forall k . k in A ==> m1 at k = m2 at k)
                  let {L1 := (!chain-> [true
                                         ==> (h in A)            [Set.in-lemma-1]
                                         ==> (m1 at h = m2 at h) [hyp]]);
                       L2 := pick-any k:'K
                               (!chain [(k in t)
                                         ==> (k in A)             [Set.in-def]
                                         ==> (m1 at k = m2 at k)       [hyp]]);
                       L3 := (!chain-> [L2 ==> (agree-on m1 m2 t) [IH]])}
                    (!chain-> [L1
                          ==> (L1 & L3)           [augment]
                          ==> (agree-on m1 m2 A)  [agree-on-def]])}
        (!equiv p1 p2)
}
```

```
628
629 define AGC := agreement-characterization
630
631 conclude downward-agreement-lemma :=
632   (forall B A m1 m2 . (agree-on m1 m2 A) & B subset A ==> (agree-on m1 m2 B))
633 pick-any B:(Set.Set 'K) A:(Set.Set 'K) m1:(DMap 'K 'V) m2:(DMap 'K 'V)
634   assume hyp := ((agree-on m1 m2 A) & B subset A)
635     let {L := pick-any k:'K
636                 assume hyp := (k in B)
637                   (!chain-> [hyp
638                         ==> (k in A) [Set.SC]
639                         ==> (m1 at k = m2 at k)  [AGC]])}
640       (!chain-> [L ==> (agree-on m1 m2 B) [AGC]])
641
642 define ms-lemma-1b := (forall m k . ~ k in dom m ==> forall v . ~ k @ v in dmap->set m)
643
644 by-induction ms-lemma-1b {
645   (m as (empty-map d:'V)) =>
646     pick-any k
647      assume hyp := (~ k in dom m)
648        pick-any v:'V
649          (!by-contradiction (~ k @ v in dmap->set m)
650             (!chain [(k @ v in dmap->set m)
651                  ==> (k @ v in Set.null)     [dmap->set-def]
652                  ==> false                    [Set.NC]]))
653 | (m as (update (pair key:'K val:'V) rest)) =>
654    let {IH :  (forall k . ~ k in dom rest ==> forall v . ~ k @ v in dmap->set rest)}
655      pick-any k
656        assume hyp := (~ k in dom m)
657          pick-any v:'V
658            (!by-contradiction (~ k @ v in dmap->set m)
659               assume sup := (k @ v in dmap->set m)
660               (!two-cases
661                assume (val = default rest)
662                  (!chain-> [sup
663                         ==> (k @ v in dmap->set rest - key)  [dmap->set-def]
664                         ==> (k in dom rest - key)            [ms-lemma-1]
665                         ==> (k in dom m)                     [dom-corrolary-1]
666                         ==> (k in dom m & hyp)               [augment]
667                         ==> false                            [prop-taut]])
668                assume (val =/= default rest)
669                let {C :=
670                      (!chain-> [sup
671                             ==> (k @ v in key @ val Set.++ dmap->set rest - key)    [dmap->set-def]
672                             ==> (k @ v = key @ val | k @ v in dmap->set rest - key)  [Set.in-def]]);
673                   _ := (!chain-> [true ==> (dom rest - key Set.subset dom rest)   [dom-lemma-3]])
674                  }
675                 (!cases C
676                   assume case1 := (k @ v = key @ val)
677                     let {L := (!chain-> [(val =/= default rest)
678                                          ==> (key in dom m)         [dom-lemma-1]])}
679                      (!chain-> [case1
680                             ==> (k = key & v = val)         [pair-axioms]
681                             ==> (k = key)                   [left-and]
682                             ==> (k in dom m)                [L]
683                             ==> (k in dom m & ~ k in dom m) [augment]
684                             ==> false                       [prop-taut]])
685                   assume case2 := (k @ v in dmap->set rest - key)
686                     (!chain-> [case2
687                             ==> (k in dom rest - key)       [ms-lemma-1]
688                             ==> (k in dom rest)             [Set.SC]
689                             ==> (k in key Set.++ dom rest) [Set.in-lemma-3]
690                             ==> (k in dom m)                [dom-def]
691                             ==> (k in dom m & ~ k in dom m) [augment]
692                             ==> false                       [prop-taut]])))))
693 }
694
695 conclude ms-lemma-1b' := (forall m k . ~ k in dom m ==> ~ exists v . k @ v in dmap->set m)
696 pick-any m:(DMap 'K 'V) k:'K
697   assume h := (~ k in dom m)
```

```
    let {p := (!chain-> [h ==> (forall v . ~ k @ v in dmap->set m) [ms-lemma-1b]])}
      (!by-contradiction (~ exists v . k @ v in dmap->set m)
        assume hyp := (exists v . k @ v in dmap->set m)
          pick-witness w for hyp wp
            (!absurd wp (!chain-> [true ==> (~ k @ w in dmap->set m) [p]])))

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

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

define (property m)  :=
  (forall k v . k @ v in dmap->set m ==> m at k = v)

define ms-theorem-1 := (forall m . property m)

(!strong-induction.measure-induction ms-theorem-1 len
    pick-any m:(DMap 'K 'V)
      assume IH := (forall m' . len m' < len m ==> property m')
        conclude (property m)
          datatype-cases (property m) on m  {
            (em as (empty-map d:'V)) =>
              pick-any k:'K v:'V
                (!chain [(k @ v in dmap->set em)
                     ==> (k @ v in Set.null)    [dmap->set-def]
                     ==> false                   [Set.NC]
                     ==> (em at k = v)           [prop-taut]])
          | (map as (update (pair key:'K val:'V) rest)) =>
              pick-any k:'K v:'V
               let {goal := (k @ v in dmap->set map ==> map at k = v);
                    lemma1 := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
                                              ==> (len rest - key < len m)   [(m = map)]]);
                    lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
                                              ==> (len rest < len m)   [(m = map)]]);
                    #lemma3 := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]]);
                    #lemma4 := (!chain-> [true ==> (dom rest subset dom map) [dom-lemma-2]]);
                    M := method (case)
                         # case here must be this assumption: (k @ v in dmap->set rest - key)
                         let {L := (!chain-> [case ==> (rest - key at k = v) [IH]]);
                              L1 := (!chain-> [case ==> (k in dom rest - key)  [ms-lemma-1]]);
                              L2 := (!by-contradiction (k =/= key)
                                      assume (k = key)
                                        (!absurd (!chain-> [true ==> (~ key in dom rest - key) [rc0]
                                                           ==> (~ k in dom rest - key)   [(k = key)]])
                                                 L1));
                              _ := (!ineq-sym L2)}
                           (!chain-> [(key =/= k)
                                 ==> (rest - key at k = rest at k)  [rc2]
                                 ==> (v = rest at k)                [L]
                                 ==> (rest at k = v)                [sym]
                                 ==> (map at k = v)                 [apply-def]])}
                (!two-cases
                 assume (val = default rest)
                   assume hyp := (k @ v in dmap->set map)
                     let {L := (!chain-> [hyp ==> (k @ v in dmap->set rest - key) [dmap->set-def]])}
                       (!M L)
                 assume (val =/= default rest)
                 assume (k @ v in dmap->set map)
                   let {D := (!chain-> [(k @ v in dmap->set map)
                                   ==> (k @ v in (key @ val) ++ dmap->set (rest - key))  [dmap->set-def]
                                   ==> (k @ v = key @ val | k @ v in dmap->set (rest - key)) [Set.in-def]])}
                  (!cases D
                    assume case1 := (k @ v = key @ val)
                      let {
                           L1 := (!chain-> [case1
                                       ==> (k = key & v = val)  [pair-axioms]]);
                           L2 := (!chain-> [(k = key) ==> (key = k) [sym]]);
                           L3 := (!chain-> [(v = val) ==> (val = v) [sym]])
```

```
                          }
                        (!chain-> [(key = k)
                                   ==> (map at k = val)    [apply-def]
                                   ==> (map at k = v)      [(val = v)]])
                     assume case2 := (k @ v in dmap->set (rest - key))
                       (!M case2)))

          })

conclude ms-theorem-2 :=
  (forall m k . ~ k in dom m ==> m at k = default m)
pick-any m:(DMap 'K 'V) k:'K
   assume hyp := (~ k in dom m)
     (!chain-> [hyp ==> (~ m at k =/= default m)  [lemma-D]
                    ==> (m at k = default m)      [dn]])

define lemma-q := (forall m k k' . k in dom m & k =/= k' ==> k in dom m - k')

by-induction lemma-q {
  (m as (empty-map d:'V)) =>
    pick-any k k'
      assume hyp := (k in dom m & k =/= k')
        (!chain-> [(k in dom m)
               ==> (k in Set.null)   [dom-def]
               ==> false              [Set.NC]
               ==> (k in dom m - k')  [prop-taut]])
| (m as (update (pair key:'K val:'V) rest)) =>
    pick-any k:'K k':'K
      assume hyp := (k in dom m & k =/= k')
       (!two-cases
        assume (val = default rest)
        let {
             _ := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]]);
             case2 := (!chain-> [(k in dom m)
                         ==> (k in dom rest - key)    [dom-def]
                         ==> (k in dom rest)          [Set.SC]]);
             IH := (forall k k' . k in dom rest & k =/= k' ==> k in dom rest - k');
             L := (!chain-> [case2
                         ==> (case2 & k =/= k')   [augment]
                         ==> (k in dom rest - k') [IH]])
           }
              (!two-cases
                assume (key = k')
                  (!chain-> [L
                         ==> (k in dom rest - key)  [(key = k')]
                         ==> (k in dom m - key)     [remove-def]
                         ==> (k in dom m - k')      [(key = k')]])
                assume (key =/= k')
                  let {_ := ();
                       p := (!chain [(dom (key @ val) ++ (rest - k'))
                                     = (key ++ dom rest - k') [dom-def]])
                       }
                    (!chain-> [L
                          ==> (k in key ++ dom rest - k')           [Set.in-lemma-3]
                          ==> (k in dom (key @ val) ++ (rest - k')) [p]
                          ==> (k in dom m - k')                     [remove-def]]))
        assume (val =/= default rest)
        let {C := (!chain-> [(k in dom m)
                         ==> (k in key ++ dom rest)    [dom-def]
                         ==> (k = key | k in dom rest) [Set.in-def]])}
          (!cases C
            assume case1 := (k = key)
              let {_ := ();
                   _ := (!chain-> [(k =/= k')
                                ==> (key =/= k')  [case1]]) ;
                   _ := (!claim (val =/= default rest));
                   L := (!chain [(dom (key @ val) ++ (rest - k'))
                                = (key ++ dom (rest - k'))  [dom-def]]);
                   ## BUG: YOU SHOULDN'T HAVE TO FORMULATE L separately here.
                   ## It should be a normal part of the following chain:
```

```
                  _ := ()
                 }
                (!chain-> [true
                      ==> (key in key ++ dom rest - k') [Set.in-lemma-1]
                      ==> (k in key ++ dom (rest - k'))   [(k = key)]
                      ==> (k in dom (key @ val) ++ (rest - k')) [L]
                      ==> (k in dom m - k')                     [remove-def]])
            assume case2 := (k in dom rest)
              let {IH := (forall k k' . k in dom rest & k =/= k' ==> k in dom rest - k');
                   L := (!chain-> [case2
                               ==> (case2 & k =/= k')   [augment]
                               ==> (k in dom rest - k') [IH]])
                 }
               (!two-cases
                 assume (key = k')
                   (!chain-> [L
                         ==> (k in dom rest - key)  [(key = k')]
                         ==> (k in dom m - key)     [remove-def]
                         ==> (k in dom m - k')      [(key = k')]])
                 assume (key =/= k')
                   let {_ := ();
                        p := (!chain [(dom (key @ val) ++ (rest - k'))
                                    = (key ++ dom rest - k') [dom-def]]);
                        # SAME PROBLEM WITH P HERE. SHOULDN'T HAVE TO DO IT
                        # SEPARATELY BY ITSELF TO USE IT IN THE CHAIN BELOW.
                        # I SHOULD BE ABLE TO SAY [DOM-DEF] IN THE STEP BELOW
                        # (RATHER THAN [P]).
                        _ := ()
                        }
                     (!chain-> [L
                          ==> (k in key ++ dom rest - k')           [Set.in-lemma-3]
                          ==> (k in dom (key @ val) ++ (rest - k')) [p]
                          ==> (k in dom m - k')                     [remove-def]])))))
}

conclude lemma-d :=
  (forall m key val . val =/= default m ==> dom key @ val ++ m = key ++ dom m - key)
pick-any m:(DMap 'K 'V) key:'K val:'V
  assume (val =/= default m)
  let {L := (dom key @ val ++ m);
       R := (key ++ dom m - key);
       R->L := (!Set.subset-intro
                  pick-any k:'K
                    assume (k in R)
                      (!cases (!chain-> [(k in R)
                                    ==> (k = key | k in dom m - key)  [Set.in-def]])
                        assume (k = key)
                          (!chain-> [true
                                ==> (key in key ++ dom m)       [Set.in-lemma-1]
                                ==> (key in dom key @ val ++ m) [dom-def]
                                ==> (k in L)                    [(k = key)]])
                        assume case2 := (k in dom m - key)
                          let {_ := (!chain-> [true ==> (dom m - key subset dom m) [dom-lemma-3]])}
                          (!chain-> [case2
                                ==> (k in dom m)        [Set.SC]
                                ==> (k in key ++ dom m) [Set.in-lemma-3]
                                ==> (k in L)               [dom-def]])));
       L->R := (!Set.subset-intro
                  pick-any k:'K
                    assume (k in L)
                       let {M := method ()
                                   (!chain-> [true
                                         ==> (key in key ++ dom m - key)  [Set.in-lemma-1]
                                         ==> (k in R)                        [(k = key)]])}
                        (!cases (!chain-> [(k in L)
                                      ==> (k in key ++ dom m)    [dom-def]
                                      ==> (k = key | k in dom m) [Set.in-def]])
                          assume (k = key)
                            (!M)
                          assume (k in dom m)
```

```
                          (!two-cases
                            assume (k = key)
                              (!M)
                            assume (k =/= key)
                              (!chain-> [(k in dom m)
                                         ==> (k in dom m & k =/= key)  [augment]
                                         ==> (k in dom m - key)        [lemma-q]
                                         ==> (k in R)                     [Set.in-def]])))))}
    (!Set.set-identity-intro L->R R->L)

define (ms-theorem-4-property m) :=
 (forall k . k in dom m ==> exists v . k @ v in dmap->set m)

define ms-theorem-4 := (forall m . ms-theorem-4-property m)

(!strong-induction.measure-induction ms-theorem-4 len
    pick-any m:(DMap 'K 'V)
      assume IH := (forall m' . len m' < len m ==> ms-theorem-4-property m')
        conclude (ms-theorem-4-property m)
           datatype-cases (ms-theorem-4-property m) on m  {
             (em as (empty-map d:'V)) =>
               pick-any k:'K
                  (!chain [(k in dom em)
                           ==> (k in Set.null)  [dom-def]
                           ==> false             [Set.NC]
                           ==> (exists v . k @ v in dmap->set em) [prop-taut]])
          | (map as (update (pair key:'K val:'V) rest)) =>
               pick-any k:'K
               let {lemma1 := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
                                              ==> (len rest - key < len m)   [(m = map)]]);
                    lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
                                              ==> (len rest < len m)   [(m = map)]]);
                   _ := ()
                  }
                assume hyp := (k in dom map)
                 (!two-cases
                  assume (val = default rest)
                    (!chain-> [hyp
                           ==> (k in dom rest - key)                      [dom-def]
                           ==> (exists v . k @ v in dmap->set rest - key) [IH]
                           ==> (exists v . k @ v in dmap->set map)        [dmap->set-def]])
                  assume (val =/= default rest)
                  (!cases (!chain-> [hyp
                                  ==> (k in key ++ dom rest - key)    [lemma-d]
                                  ==> (k = key | k in dom rest - key) [Set.in-def]])
                    assume case1 := (k = key)
                      (!chain-> [true
                             ==> (key @ val in key @ val ++ dmap->set rest - key)  [Set.in-lemma-1]
                             ==> (key @ val in dmap->set map)                      [dmap->set-def]
                             ==> (exists v . key @ v in dmap->set map)             [existence]
                             ==> (exists v . k @ v in dmap->set map)               [case1]])
                    assume case2 := (k in dom rest - key)
                      (!chain-> [case2
                             ==> (exists v . k @ v in dmap->set rest - key) [IH]
                             ==> (exists v . k @ v in key @ val ++ dmap->set rest - key) [Set.in-lemma-3]
                             ==> (exists v . k @ v in dmap->set map)                     [dmap->set-def]]))))
})

conclude at-characterization-1 :=
(forall m k v . m at k = v ==> k @ v in dmap->set m | ~ k in dom m & v = default m)
  pick-any m:(DMap 'K 'V) k:'K v:'V
    assume hyp := (m at k = v)
      (!two-cases
        assume case1 := (k in dom m)
          pick-witness val for (!chain-> [(k in dom m)
                                          ==> (exists v . k @ v in dmap->set m) [ms-theorem-4]])
           # we now have (k @ val in dmap->set m)
           let {v=val := (!chain-> [(k @ val in dmap->set m)
                                    ==> (m at k = val)           [ms-theorem-1]
                                    ==> (v = val)                [hyp]])}
```

```
978               (!chain-> [(k @ val in dmap->set m)
979                     ==> (k @ v in dmap->set m)     [v=val]
980                     ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]])
981         assume case2 := (~ k in dom m)
982           (!chain-> [case2
983                 ==> (m at k = default m)                                  [ms-theorem-2]
984                 ==> (v = default m)                                       [hyp]
985                 ==> (~ k in dom m & v = default m)                        [augment]
986                 ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]]))
987
988 conclude at-characterization-2 :=
989 (forall m k v . k @ v in dmap->set m | ~ k in dom m & v = default m ==> m at k = v)
990   pick-any m:(DMap 'K 'V) k:'K v:'V
991     assume hyp := (k @ v in dmap->set m | ~ k in dom m & v = default m)
992       (!cases hyp
993         assume case1 := (k @ v in dmap->set m)
994           (!chain-> [case1 ==> (m at k = v)   [ms-theorem-1]])
995         assume case2 := (~ k in dom m & v = default m)
996           (!chain-> [(~ k in dom m)
997                 ==> (m at k = default m)   [ms-theorem-2]
998                 ==> (m at k = v)           [(v = default m)]]))
999
1000 conclude at-characterization :=
1001 (forall m k v . m at k = v <==> k @ v in dmap->set m | ~ k in dom m & v = default m)
1002   pick-any m:(DMap 'K 'V) k:'K v:'V
1003     (!equiv
1004       (!chain [(m at k = v) ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [at-characterization-1]])
1005       (!chain [(k @ v in dmap->set m | ~ k in dom m & v = default m) ==> (m at k = v) [at-characterization-2]]))
1006
1007 define at-characterization-lemma :=
1008   (forall m k v . m at k = v & k in dom m ==> k @ v in dmap->set m)
1009
1010 define at-characterization-lemma-2 :=
1011   (forall m k v . m at k = v & v =/= default m ==> k @ v in dmap->set m)
1012
1013 (!force at-characterization-lemma)
1014 (!force at-characterization-lemma-2)
1015
1016 } # close module DMap
```