

## lib/basic/dmaps.ath

```

1 # Module for rudimentary finite maps with default values. This module
2 # is natively understood by the SMT translator, and it's how Athena handles
3 # SMT problems involving finite functions.
4
5 load "sets"
6 load "strong-induction"
7
8 module DMap {
9
10 define [null ++ in subset proper-subset \/ /\ \ card A B C] :=
11   [Set.null Set.++ Set.in Set.subset Set.proper-subset
12    Set.\ / Set.\ / Set.\ Set.card
13    ?A:(Set.Set 'S1) ?B:(Set.Set 'S2) ?C:(Set.Set 'S3)]
14
15 structure (DMap S T) := (empty-map T) | (update (Pair S T) (DMap S T))
16
17 set-precedence empty-map 250
18
19 define (alist->dmap-general L preprocessor) :=
20   match L {
21     [d (some-list pairs)] =>
22       letrec {loop := lambda (L)
23         match L {
24           [] => (empty-map d)
25           | (list-of (|| [x --> n] [x n]) rest) =>
26             (update (pair (preprocessor x) (preprocessor n)) (loop rest))}
27       (loop pairs)
28   | _ => L
29   }
30
31 define (alist->dmap L) := (alist->dmap-general L id)
32
33 define (dmap->alist-general m preprocessor) :=
34   letrec {loop := lambda (m pairs)
35     match m {
36       (empty-map d) => [d (rev pairs)]
37       | (update (pair k v) rest) =>
38         (loop rest (add [(preprocessor k) --> (preprocessor v)] pairs))
39       | _ => m}}
40   (loop m [])
41
42 (define (remove-from m k)
43   (match m
44     ((empty-map _) m)
45     ((update (binding as (pair key val)) rest)
46       (check ((equal? k key) (remove-from rest k))
47         (else (update binding (remove-from rest k)))))))
48
49 define (dmap->alist-canonical-general m preprocessor) :=
50   letrec {loop := lambda (m pairs)
51     match m {
52       (empty-map d) => [d (rev pairs)]
53       | (update (pair k v) rest) =>
54         (loop (remove-from rest k)
55           (add [(preprocessor k) --> (preprocessor v)] pairs))
56       | _ => m}}
57   (loop m [])
58
59 define (dmap->alist m) := (dmap->alist-general m id)
60
61 expand-input update [(alist->pair id id) alist->dmap]
62
63 declare apply: (K, V) [(DMap K V) K] -> V [110 [alist->dmap id]]
64
65 define [at] := [apply]
66
67 overload ++ update

```

```

68
69 set-precedence ++ 210
70
71 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]
72 define [h t] := [Set.h Set.t]
73
74 define [m m' m1 m2 rest] := [?m:(DMap 'S1 'S2) ?m':(DMap 'S3 'S4) ?m1:(DMap 'S5 'S6) ?m2:(DMap 'S7 'S8) ?rest:(DMap 'S9 'S10)]
75
76 assert* apply-def :=
77   [(empty-map d at _ = d)
78    (k @ v ++ rest at x = v <== k = x)
79    (k @ v ++ rest at x = rest at x <== k =/= x)]
80
81 ## Some testing:
82
83 define make-map :=
84   lambda (L)
85     match L {
86       [] => (empty-map 0)
87       | (list-of [x n] rest) => (update (x @ n) (make-map rest))
88     }
89
90 define update* :=
91   lambda (fm pairs)
92     letrec {loop := lambda (pairs res)
93       match pairs {
94         [] => res
95         | (list-of [key val] more) => (loop more (update res key val))}
96     (loop pairs fm)
97
98
99 define f := lambda (i) [(string->id ("s" joined-with (val->string i))) i]
100
101 define L := (from-to 1 5)
102
103 define sample-map := (make-map (map f L))
104
105 # So sample-map maps 's1 to 1, ..., 's5 to 5.
106
107 define applied-to := apply
108
109 (eval sample-map at 's1)
110 (eval sample-map at 's2)
111 (eval sample-map at 's3)
112 (eval sample-map at 's4)
113 (eval sample-map at 's5)
114
115 # And this should give the default value 0:
116
117 (eval sample-map at 's99)
118
119 declare default: (K, V) [(DMap K V)] -> V [200 [alist->dmap]]
120
121 assert* default-def :=
122   [(default empty-map d = d)
123    (default _ ++ rest = default rest)]
124
125 (eval default sample-map)
126
127 declare remove: (S, T) [(DMap S T) S] -> (DMap S T) [- 120 [alist->dmap id]]
128
129 left-assoc -
130
131 assert* remove-def :=
132   [(empty-map d - _ = empty-map d)
133    ([key _] ++ rest - key = rest - key)
134    (key =/= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]
135
136 declare dom: (S, T) [(DMap S T)] -> (Set.Set S) [[alist->dmap]]
137

```

```

138 assert* dom-def :=
139   [(dom empty-map _ = null)
140    (dom [k v] ++ rest = dom rest - k <== v = default rest)
141    (dom [k v] ++ rest = k ++ dom rest <== v /= default rest)]
142
143 declare size: (S, T) [(DMap S T)] -> N [[alist->dmap]]
144 assert* size-axioms := [(size m = card dom m)]
145
146 define rc1 := (forall m x . (m - x) at x = default m)
147
148 by-induction rc1 {
149   (m as (empty-map d)) =>
150     pick-any x
151       (!chain [(m - x) at x]
152              = (m at x) [remove-def]
153              = d [apply-def]
154              = (default m) [default-def]))
155 | (m as (update (pair k:'S v) rest)) =>
156   let {IH := (forall x . rest - x at x = default rest)}
157     pick-any x:'S
158       (!two-cases
159         assume (k = x)
160           (!chain [(m - x) at x]
161                  = (m - k at k) [(k = x)]
162                  = (rest - k at k) [remove-def]
163                  = (default rest) [IH]
164                  = (default m) [default-def]
165                  ])
166         assume (k /= x)
167           (!chain [(m - x) at x]
168                  = ((k @ v) ++ (rest - x) at x) [remove-def]
169                  = (rest - x at x) [apply-def]
170                  = (default rest) [IH]
171                  = (default m) [default-def])))
172 }
173
174 define rc2 := (forall m k x . k /= x ==> m - k at x = m at x)
175
176 by-induction rc2 {
177   (m as (empty-map d:'V)) =>
178     pick-any k:'K x:'K
179       assume (k /= x)
180       let {L := (m - k at x);
181           R := (m at x)}
182         (!chain [L
183                 = (m at x) [remove-def]])
184 | (m as (update (pair key:'K val:'V) rest:(DMap 'K 'V))) =>
185   pick-any k:'K x:'K
186     assume (k /= x)
187     let {IH := (forall k x . k /= x ==> (rest - k) at x = rest at x)}
188       (!two-cases
189         assume (key = k)
190         let {_ := (!by-contradiction (key /= x)
191                                     (!chain [(key = x)
192                                             ==> (k = x) [(key = k)]
193                                             ==> (k = x & k /= x) [augment]
194                                             ==> false [prop-taut]])})
195           (!chain [(m - k) at x]
196                  = (((k @ val) ++ rest) - k at x) [(key = k)]
197                  = (rest - k at x) [remove-def]
198                  = (rest at x) [IH]
199                  = (m at x) [apply-def]))
200         assume (key /= k)
201           (!two-cases
202             assume (x = key)
203               (!chain [(m - k) at x]
204                      = (([key val] ++ (rest - k)) at x) [remove-def]
205                      = (([x val] ++ (rest - k)) at x) [(x = key)]
206                      = val [apply-def]
207                      = (([x val] ++ rest) at x) [apply-def])

```

```

208         = (m at x)                                [(x = key)]]
209     assume (x != key)
210     (!chain [(m - k at x)
211             = (([key val] ++ (rest - k)) at x) [remove-def]
212             = (rest - k at x)                 [apply-def]
213             = (rest at x)                     [IH]
214             = (m at x)                       [apply-def]]))
215 }
216
217 define rc3 := (forall m k . default m = default m - k)
218 by-induction rc3 {
219   (m as (empty-map d:'V)) =>
220     pick-any k
221     (!chain [(default m)
222             = (default m - k) [remove-def]])
223 | (m as (update (pair key:'K val:'V) rest)) =>
224   let {IH := (forall k . default rest = default rest - k)}
225     pick-any k:'K
226     (!two-cases
227       assume (key = k)
228       (!combine-equations
229         (!chain [(default m)
230                 = (default rest)           [default-def]
231                 = (default rest - k)      [IH]])
232         (!chain [(default m - k)
233                 = (default rest - k)      [remove-def]]))
234       assume (key != k)
235       (!chain-> [(default m - k)
236                = (default key @ val ++ rest - k) [remove-def]
237                = (default rest - k)             [default-def]
238                = (default rest)                 [IH]
239                = (default m)                   [default-def]
240                ==> (default m - k = default m)
241                ==> (default m = default m - k)  [sym]]))
242 }
243
244 conclude dom-lemma-1 :=
245   (forall k v rest . v != default rest ==> k in dom [k v] ++ rest)
246 pick-any k v rest
247   assume hyp := (v != default rest)
248   (!chain-> [true ==> (k in k ++ dom rest)      [Set.in-lemma-1]
249             ==> (k in dom [k v] ++ rest) [dom-def]])
250
251 conclude dom-lemma-2 :=
252   (forall m k v . v != default m ==> dom m subset dom [k v] ++ m)
253 pick-any m k v
254   assume hyp := (v != default m)
255   (!Set.subset-intro
256     pick-any x
257     (!chain [(x in dom m)
258             ==> (x in k ++ dom m)           [Set.in-lemma-3]
259             ==> (x in dom [k v] ++ m)      [dom-def]]))
260
261 conclude dom-lemma-2b :=
262   (forall m x k v . v != default m & x in dom m ==> x in dom [k v] ++ m)
263 pick-any m x k v
264   assume (v != default m & x in dom m)
265   let {_ := (!chain-> [(v != default m) ==> (dom m subset dom [k v] ++ m) [dom-lemma-2]])}
266   (!chain-> [(x in dom m) ==> (x in dom [k v] ++ m) [Set.SC]])
267
268 # conclude dom-lemma-2c :=
269 #   (forall m x k v . x in dom [k v] ++ m ==> x = k | x in dom m - k)
270 # pick-any m: (DMap 'K 'V) x:'K k:'K v:'V
271 #   assume hyp := (x in dom [k v] ++ m)
272 #   (!two-cases
273 #     assume (v = default m)
274 #     (!chain-> [hyp
275                ==> (x in dom m - k)           [dom-def]
276                ==> (x = k | x in dom m - k) [prop-taut]])
277 #     assume (v != default m)

```

```

278 #           (!chain-> [hyp
279 #               ==> (x in k ++ dom m)           [dom-def]
280 #               ==> (x = k | x in dom m - k)     [Set.in-def]]))
281
282 define [< <=] := [N.< N.<=]
283 declare len: (S, T) [(DMap S T)] -> N [[alist->dmap]]
284
285 assert* len-def :=
286   [(len empty-map _ = zero)
287    (len _ @ _ ++ rest = S len rest)]
288
289 define len-lemma-1 :=
290   (forall m k v . len m < len (k @ v) ++ m)
291
292 by-induction len-lemma-1 {
293   (m as (empty-map d:'V)) =>
294     pick-any k v
295     let {len-left := (!chain [(len m) = zero           [len-def]]);
296         len-right := (!chain [(len k @ v ++ m) = (S len m) [len-def]]);
297         (!chain-> [true
298                   ==> (zero < S len m)           [N.Less.<-def]
299                   ==> (len m < len k @ v ++ m) [len-left len-right])
300   | (m as (update (pair key:'K val:'V) rest)) =>
301     let {IH := (forall k v . len rest < len k @ v ++ rest)}
302         pick-any k:'K v:'V
303         let {len-left := (!chain [(len m)
304                                   = (S len rest) [len-def]]);
305             len-right := (!chain [(len k @ v ++ m)
306                                   = (S len m)      [len-def]
307                                   = (S S len rest) [len-left])];
308         (!chain-> [true
309                   ==> (S len rest < S S len rest) [N.Less.<S]
310                   ==> (len m < len k @ v ++ m)   [len-left len-right])
311   }
312
313 conclude len-lemma-2 := (forall m k . len m - k <= len m)
314 by-induction len-lemma-2 {
315   (m as (empty-map d:'V)) =>
316     pick-any k
317     (!chain-> [(len m - k)
318               = (len m)           [remove-def]
319               ==> (len m - k <= len m) [N.Less.=.<=-def]])
320   | (m as (update (pair key:'K val:'V) rest)) =>
321     pick-any k:'K
322     let {IH := (forall k . len rest - k <= len rest);
323         L2 := (!chain-> [true ==> (len rest - k <= len rest) [IH]]);
324         L3 := (!chain-> [true ==> (len rest < len m)           [len-lemma-1]]);
325         L4 := (!chain-> [L2 ==> (L2 & L3)                       [augment]
326                         ==> (len rest - k < len m)           [N.Less=.transitive2]})
327     (!two-cases
328       assume (key = k)
329         (!chain-> [(len m - k)
330                   = (len rest - k)           [remove-def]
331                   ==> (len m - k <= len rest - k) [N.Less.=.<=-def]
332                   ==> (len m - k <= len rest - k & L2) [augment]
333                   ==> (len m - k <= len rest)         [N.Less=.transitive]
334                   ==> (len m - k <= len rest & L3)     [augment]
335                   ==> (len m - k < len m)             [N.Less=.transitive2]
336                   ==> (len m - k <= len m)           [N.Less.=.<=-def]])
337       assume (key /= k)
338         let {L5 := (!chain-> [(len m - k)
339                               = (len [key val] ++ (rest - k)) [remove-def]
340                               = (S len rest - k)             [len-def])];
341         (!chain-> [L4
342                   ==> (S len rest - k <= len m) [N.Less=.discrete]
343                   ==> (len m - k <= len m)     [L5])])
344   }
345
346
347 define len-lemma-3 :=

```



```

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

```





```

558         ==> (k @ v in dmap->set rest - key) [dmap->set-def]
559         ==> (k in dom rest - key) [IH]
560         ==> (k in dom map) [dom-def]]
561     assume C2 := (val != default rest)
562     let {_ := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]])}
563     (!chain [(k @ v in dmap->set map)
564             ==> (k @ v in key @ val ++ dmap->set rest - key) [dmap->set-def]
565             ==> (k @ v = key @ val | k @ v in dmap->set rest - key) [Set.in-def]
566             ==> (k = key & v = val | k @ v in dmap->set rest - key) [pair-axioms]
567             ==> (k = key | k @ v in dmap->set rest - key) [prop-taut]
568             ==> (k = key | k in dom rest - key) [IH]
569             ==> (k = key | k in dom rest) [Set.SC]
570             ==> (k in key ++ dom rest) [Set.in-def]
571             ==> (k in dom map) [dom-def]])
572     )
573   })
574
575 # conclude dom-corrolary-1 :=
576 # (forall key val k rest . k in dom rest - key ==> k in dom [key val] ++ rest)
577 # pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
578 # let {l1 := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]])}
579 # (!two-cases
580 #   assume (val = default rest)
581 #   (!chain [(k in dom rest - key)
582 #     ==> (k in dom [key val] ++ rest) [dom-def]])
583 #   assume (val != default rest)
584 #   (!chain [(k in dom rest - key)
585 #     ==> (k in dom rest) [Set.SC]
586 #     ==> (k = key | k in dom rest) [prop-taut]
587 #     ==> (k in key ++ dom rest) [Set.in-def]
588 #     ==> (k in dom [key val] ++ rest) [dom-def])])
589
590 assert* dmap-identity :=
591 (forall m1 m2 . m1 = m2 <==> default m1 = default m2 & dmap->set m1 = dmap->set m2)
592
593 define dmap-identity-characterization :=
594 (forall m1 m2 . m1 = m2 <==> forall k . m1 at k = m2 at k)
595
596 declare agree-on: (S, T) [(DMap S T) (DMap S T) (Set.Set S)] -> Boolean
597 [[alist->dmap alist->dmap Set.lst->set]]
598
599
600 assert* agree-on-def :=
601 [(agree-on m1 m2 null)
602  ((agree-on m1 m2 h Set.++ t) <==> m1 at h = m2 at h & (agree-on m1 m2 t))]
603
604 let {m1 := [77 [['x --> 1] ['y --> 2]]};
605     m2 := [78 [['y --> 2] ['x --> 1]]];
606 (eval (agree-on m1 m2 ['x 'y]))
607
608 define agreement-characterization :=
609 (forall A m1 m2 . (agree-on m1 m2 A) <==> forall k . k in A ==> m1 at k = m2 at k)
610
611 by-induction agreement-characterization {
612 (A as Set.null:(Set.Set 'K)) =>
613   pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
614     let {p1 := assume (agree-on m1 m2 A)
615         pick-any k:'K
616           (!chain [(k in A)
617                   ==> false [Set.NC]
618                   ==> (m1 at k = m2 at k) [prop-taut]]);
619         p2 := assume (forall k . k in A ==> m1 at k = m2 at k)
620           (!chain-> [true ==> (agree-on m1 m2 A) [agree-on-def]])}
621   (!equiv p1 p2)
622 | (A as (Set.insert h:'K t:(Set.Set 'K))) =>
623   let {IH := (forall m1 m2 . (agree-on m1 m2 t) <==> forall k . k in t ==> m1 at k = m2 at k)}
624   pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
625     let {p1 := assume hyp := (agree-on m1 m2 A)
626         pick-any k:'K
627         assume (k in A)

```

```

628         (!cases (!chain-> [(k in A)
629                             ==> (k = h | k in t)   [Set.in-def]]))
630         assume (k = h)
631         (!chain-> [hyp
632                   ==> (m1 at h = m2 at h)         [agree-on-def]
633                   ==> (m1 at k = m2 at k)         [(k = h)]]))
634         assume (k in t)
635         let {P := (!chain-> [hyp
636                           ==> (agree-on m1 m2 t)   [agree-on-def]
637                           ==> (forall k . k in t ==> m1 at k = m2 at k) [IH])]}
638         (!chain-> [(k in t) ==> (m1 at k = m2 at k) [P]]);
639 p2 := assume hyp := (forall k . k in A ==> m1 at k = m2 at k)
640         let {L1 := (!chain-> [true
641                             ==> (h in A)           [Set.in-lemma-1]
642                             ==> (m1 at h = m2 at h) [hyp]]);
643          L2 := pick-any k:'K
644                (!chain [(k in t)
645                          ==> (k in A)             [Set.in-def]
646                          ==> (m1 at k = m2 at k)   [hyp]]);
647          L3 := (!chain-> [L2 ==> (agree-on m1 m2 t) [IH]])}
648         (!chain-> [L1
649                   ==> (L1 & L3)                   [augment]
650                   ==> (agree-on m1 m2 A)          [agree-on-def]]))
651         (!equiv p1 p2)
652     }
653
654 define AGC := agreement-characterization
655
656 conclude downward-agreement-lemma :=
657     (forall B A m1 m2 . (agree-on m1 m2 A) & B subset A ==> (agree-on m1 m2 B))
658 pick-any B:(Set.Set 'K) A:(Set.Set 'K) m1:(DMap 'K 'V) m2:(DMap 'K 'V)
659     assume hyp := ((agree-on m1 m2 A) & B subset A)
660     let {L := pick-any k:'K
661           assume hyp := (k in B)
662           (!chain-> [hyp
663                     ==> (k in A) [Set.SC]
664                     ==> (m1 at k = m2 at k) [AGC]])}
665         (!chain-> [L ==> (agree-on m1 m2 B) [AGC]])
666
667 define ms-lemma-1b := (forall m k . ~ k in dom m ==> forall v . ~ k @ v in dmap->set m)
668
669 by-induction ms-lemma-1b {
670   (m as (empty-map d:'V)) =>
671     pick-any k
672     assume hyp := (~ k in dom m)
673     pick-any v:'V
674     (!by-contradiction (~ k @ v in dmap->set m)
675       (!chain [(k @ v in dmap->set m)
676               ==> (k @ v in Set.null) [dmap->set-def]
677               ==> false [Set.NC]]))
678 | (m as (update (pair key:'K val:'V) rest)) =>
679   let {IH := (forall k . ~ k in dom rest ==> forall v . ~ k @ v in dmap->set rest)}
680     pick-any k
681     assume hyp := (~ k in dom m)
682     pick-any v:'V
683     (!by-contradiction (~ k @ v in dmap->set m)
684       assume sup := (k @ v in dmap->set m)
685       (!two-cases
686         assume (val = default rest)
687         (!chain-> [sup
688                   ==> (k @ v in dmap->set rest - key) [dmap->set-def]
689                   ==> (k in dom rest - key) [ms-lemma-1]
690                   ==> (k in dom m) [dom-corrolary-1]
691                   ==> (k in dom m & hyp) [augment]
692                   ==> false [prop-taut]])
693         assume (val /= default rest)
694         let {C :=
695               (!chain-> [sup
696                         ==> (k @ v in key @ val Set.++ dmap->set rest - key) [dmap->set-def]
697                         ==> (k @ v = key @ val | k @ v in dmap->set rest - key) [Set.in-def]]);

```

```

698     _ := (!chain-> [true ==> (dom rest - key Set.subset dom rest) [dom-lemma-3]])
699   }
700   (!cases C
701     assume case1 := (k @ v = key @ val)
702     let {L := (!chain-> [(val != default rest)
703                       ==> (key in dom m) [dom-lemma-1]])}
704     (!chain-> [case1
705               ==> (k = key & v = val) [pair-axioms]
706               ==> (k = key) [left-and]
707               ==> (k in dom m) [L]
708               ==> (k in dom m & ~ k in dom m) [augment]
709               ==> false [prop-taut]])
710     assume case2 := (k @ v in dmap->set rest - key)
711     (!chain-> [case2
712               ==> (k in dom rest - key) [ms-lemma-1]
713               ==> (k in dom rest) [Set.SC]
714               ==> (k in key Set.++ dom rest) [Set.in-lemma-3]
715               ==> (k in dom m) [dom-def]
716               ==> (k in dom m & ~ k in dom m) [augment]
717               ==> false [prop-taut]]))
718   }
719
720 conclude ms-lemma-1b' := (forall m k . ~ k in dom m ==> ~ exists v . k @ v in dmap->set m)
721 pick-any m: (DMap 'K 'V) k:'K
722   assume h := (~ k in dom m)
723   let {p := (!chain-> [h ==> (forall v . ~ k @ v in dmap->set m) [ms-lemma-1b']]}
724     (!by-contradiction (~ exists v . k @ v in dmap->set m)
725       assume hyp := (exists v . k @ v in dmap->set m)
726       pick-witness w for hyp wp
727       (!absurd wp (!chain-> [true ==> (~ k @ w in dmap->set m) [p]]))
728     )
729 declare restricted-to: (S, T) [(DMap S T) (Set.Set S)] -> (DMap S T) [150 | ^ [alist->dmap Set.lst->set]]
730
731 assert* restrict-axioms :=
732   [(empty-map d | ^ _ = empty-map d)
733    (k in A ==> [k v] ++ rest | ^ A = [k v] ++ (rest | ^ A))
734    (~ k in A ==> [k v] ++ rest | ^ A = rest | ^ A)]
735
736 define sm1 := [0 [['x --> 1] ['y --> 2] ['z --> 3]]]
737 define sm2 := [0 [['y --> 2] ['z --> 3] ['x --> 1]]]
738
739 (eval sm1 | ^ ['z 'y])
740
741 define (property m) :=
742   (forall k v . k @ v in dmap->set m ==> m at k = v)
743
744 define ms-theorem-1 := (forall m . property m)
745
746 (!strong-induction.measure-induction ms-theorem-1 len
747   pick-any m: (DMap 'K 'V)
748     assume IH := (forall m' . len m' < len m ==> property m')
749     conclude (property m)
750     datatype-cases (property m) on m {
751       (em as (empty-map d:'V)) =>
752         pick-any k:'K v:'V
753         (!chain [(k @ v in dmap->set em)
754                 ==> (k @ v in Set.null) [dmap->set-def]
755                 ==> false [Set.NC]
756                 ==> (em at k = v) [prop-taut]])
757       | (map as (update (pair key:'K val:'V) rest)) =>
758         pick-any k:'K v:'V
759         let {goal := (k @ v in dmap->set map ==> map at k = v);
760              lemmal := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
761                                   ==> (len rest - key < len m) [(m = map)]]);
762              lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
763                                   ==> (len rest < len m) [(m = map)]]);
764              #lemma3 := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]]);
765              #lemma4 := (!chain-> [true ==> (dom rest subset dom map) [dom-lemma-2]]);
766              M := method (case)
767                # case here must be this assumption: (k @ v in dmap->set rest - key)

```

```

768     let {L := (!chain-> [case ==> (rest - key at k = v) [IH]]);
769         L1 := (!chain-> [case ==> (k in dom rest - key) [ms-lemma-1]]);
770         L2 := (!by-contradiction (k /= key)
771             assume (k = key)
772                 (!absurd (!chain-> [true ==> (~ key in dom rest - key) [rc0]
773                     ==> (~ k in dom rest - key) [(k = key)]])
774                     L1));
775         _ := (!ineq-sym L2)}
776     (!chain-> [(key /= k)
777         ==> (rest - key at k = rest at k) [rc2]
778         ==> (v = rest at k) [L]
779         ==> (rest at k = v) [sym]
780         ==> (map at k = v) [apply-def]]})
781     (!two-cases
782     assume (val = default rest)
783     assume hyp := (k @ v in dmap->set map)
784     let {L := (!chain-> [hyp ==> (k @ v in dmap->set rest - key) [dmap->set-def]]}
785         (!M L)
786     assume (val /= default rest)
787     assume (k @ v in dmap->set map)
788     let {D := (!chain-> [(k @ v in dmap->set map)
789         ==> (k @ v in (key @ val) ++ dmap->set (rest - key)) [dmap->set-def]
790         ==> (k @ v = key @ val | k @ v in dmap->set (rest - key)) [Set.in-def]]})
791     (!cases D
792     assume case1 := (k @ v = key @ val)
793     let {
794         L1 := (!chain-> [case1
795             ==> (k = key & v = val) [pair-axioms]]);
796         L2 := (!chain-> [(k = key) ==> (key = k) [sym]]);
797         L3 := (!chain-> [(v = val) ==> (val = v) [sym]]);
798     }
799     (!chain-> [(key = k)
800         ==> (map at k = val) [apply-def]
801         ==> (map at k = v) [(val = v)]])
802     assume case2 := (k @ v in dmap->set (rest - key))
803         (!M case2))
804
805     })
806
807     conclude ms-theorem-2 :=
808     (forall m k . ~ k in dom m ==> m at k = default m)
809     pick-any m: (DMap 'K 'V) k:'K
810     assume hyp := (~ k in dom m)
811     (!chain-> [hyp ==> (~ m at k /= default m) [lemma-D]
812         ==> (m at k = default m) [dn]])
813
814     define lemma-q := (forall m k k' . k in dom m & k /= k' ==> k in dom m - k')
815
816     by-induction lemma-q {
817     (m as (empty-map d:'V)) =>
818     pick-any k k'
819     assume hyp := (k in dom m & k /= k')
820     (!chain-> [(k in dom m)
821         ==> (k in Set.null) [dom-def]
822         ==> false [Set.NC]
823         ==> (k in dom m - k') [prop-taut]])
824     | (m as (update (pair key:'K val:'V) rest)) =>
825     pick-any k:'K k':'K
826     assume hyp := (k in dom m & k /= k')
827     (!two-cases
828     assume (val = default rest)
829     let {
830         _ := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]]);
831         case2 := (!chain-> [(k in dom m)
832             ==> (k in dom rest - key) [dom-def]
833             ==> (k in dom rest) [Set.SC]]);
834         IH := (forall k k' . k in dom rest & k /= k' ==> k in dom rest - k');
835         L := (!chain-> [case2
836             ==> (case2 & k /= k') [augment]
837             ==> (k in dom rest - k') [IH]])

```

```

838     }
839     (!two-cases
840       assume (key = k')
841       (!chain-> [L
842         ==> (k in dom rest - key) [(key = k')]
843         ==> (k in dom m - key) [remove-def]
844         ==> (k in dom m - k') [(key = k')]])
845       assume (key /= k')
846       let {_ := ()};
847       p := (!chain [(dom (key @ val) ++ (rest - k'))
848         = (key ++ dom rest - k') [dom-def]])
849       }
850       (!chain-> [L
851         ==> (k in key ++ dom rest - k') [Set.in-lemma-3]
852         ==> (k in dom (key @ val) ++ (rest - k')) [p]
853         ==> (k in dom m - k') [remove-def]]))
854     assume (val /= default rest)
855     let {C := (!chain-> [(k in dom m)
856       ==> (k in key ++ dom rest) [dom-def]
857       ==> (k = key | k in dom rest) [Set.in-def]])}
858     (!cases C
859       assume case1 := (k = key)
860       let {_ := ()};
861       _ := (!chain-> [(k /= k')
862         ==> (key /= k') [case1]]);
863       _ := (!claim (val /= default rest));
864       L := (!chain [(dom (key @ val) ++ (rest - k'))
865         = (key ++ dom (rest - k')) [dom-def]]);
866       ## BUG: YOU SHOULDN'T HAVE TO FORMULATE L separately here.
867       ## It should be a normal part of the following chain:
868       _ := ()
869       }
870       (!chain-> [true
871         ==> (key in key ++ dom rest - k') [Set.in-lemma-1]
872         ==> (k in key ++ dom (rest - k')) [(k = key)]
873         ==> (k in dom (key @ val) ++ (rest - k')) [L]
874         ==> (k in dom m - k') [remove-def]])
875       assume case2 := (k in dom rest)
876       let {IH := (forall k k' . k in dom rest & k /= k' ==> k in dom rest - k');
877         L := (!chain-> [case2
878           ==> (case2 & k /= k') [augment]
879           ==> (k in dom rest - k') [IH]]
880         )}
881       (!two-cases
882         assume (key = k')
883         (!chain-> [L
884           ==> (k in dom rest - key) [(key = k')]
885           ==> (k in dom m - key) [remove-def]
886           ==> (k in dom m - k') [(key = k')]])
887         assume (key /= k')
888         let {_ := ()};
889         p := (!chain [(dom (key @ val) ++ (rest - k'))
890           = (key ++ dom rest - k') [dom-def]]);
891         # SAME PROBLEM WITH P HERE. SHOULDN'T HAVE TO DO IT
892         # SEPARATELY BY ITSELF TO USE IT IN THE CHAIN BELOW.
893         # I SHOULD BE ABLE TO SAY [DOM-DEF] IN THE STEP BELOW
894         # (RATHER THAN [P]).
895         _ := ()
896         }
897         (!chain-> [L
898           ==> (k in key ++ dom rest - k') [Set.in-lemma-3]
899           ==> (k in dom (key @ val) ++ (rest - k')) [p]
900           ==> (k in dom m - k') [remove-def]]))
901     }
902
903 conclude lemma-d :=
904   (forall m key val . val /= default m ==> dom key @ val ++ m = key ++ dom m - key)
905 pick-any m: (DMap 'K 'V) key:'K val:'V
906 assume (val /= default m)
907 let {L := (dom key @ val ++ m);

```

```

908   R := (key ++ dom m - key);
909   R->L := (!Set.subset-intro
910     pick-any k:'K
911     assume (k in R)
912     (!chain-> [(k in R)
913       ==> (k = key | k in dom m - key) [Set.in-def]])
914     assume (k = key)
915     (!chain-> [true
916       ==> (key in key ++ dom m) [Set.in-lemma-1]
917       ==> (key in dom key @ val ++ m) [dom-def]
918       ==> (k in L) [(k = key)]])
919     assume case2 := (k in dom m - key)
920     let {_ := (!chain-> [true ==> (dom m - key subset dom m) [dom-lemma-3]])}
921     (!chain-> [case2
922       ==> (k in dom m) [Set.SC]
923       ==> (k in key ++ dom m) [Set.in-lemma-3]
924       ==> (k in L) [dom-def]]));
925   L->R := (!Set.subset-intro
926     pick-any k:'K
927     assume (k in L)
928     let {M := method ()
929       (!chain-> [true
930         ==> (key in key ++ dom m - key) [Set.in-lemma-1]
931         ==> (k in R) [(k = key)]])}
932     (!cases (!chain-> [(k in L)
933       ==> (k in key ++ dom m) [dom-def]
934       ==> (k = key | k in dom m) [Set.in-def]])
935     assume (k = key)
936     (!M)
937     assume (k in dom m)
938     (!two-cases
939       assume (k = key)
940       (!M)
941       assume (k /= key)
942       (!chain-> [(k in dom m)
943         ==> (k in dom m & k /= key) [augment]
944         ==> (k in dom m - key) [lemma-q]
945         ==> (k in R) [Set.in-def]])))))
946   (!Set.set-identity-intro L->R R->L)
947
948   define (ms-theorem-4-property m) :=
949     (forall k . k in dom m ==> exists v . k @ v in dmap->set m)
950
951   define ms-theorem-4 := (forall m . ms-theorem-4-property m)
952
953   (!strong-induction.measure-induction ms-theorem-4 len
954     pick-any m:(DMap 'K 'V)
955     assume IH := (forall m' . len m' < len m ==> ms-theorem-4-property m')
956     conclude (ms-theorem-4-property m)
957     datatype-cases (ms-theorem-4-property m) on m {
958       (em as (empty-map d:'V)) =>
959         pick-any k:'K
960         (!chain [(k in dom em)
961           ==> (k in Set.null) [dom-def]
962           ==> false [Set.NC]
963           ==> (exists v . k @ v in dmap->set em) [prop-taut]])
964       | (map as (update (pair key:'K val:'V) rest)) =>
965         pick-any k:'K
966         let {lemma1 := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
967           ==> (len rest - key < len m) [(m = map)]]);
968           lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
969           ==> (len rest < len m) [(m = map)]]);
970         _ := ()
971       }
972     assume hyp := (k in dom map)
973     (!two-cases
974       assume (val = default rest)
975       (!chain-> [hyp
976         ==> (k in dom rest - key) [dom-def]
977         ==> (exists v . k @ v in dmap->set rest - key) [IH]

```

```

978         ==> (exists v . k @ v in dmap->set map)          [dmap->set-def]])
979   assume (val != default rest)
980   (!cases (!chain-> [hyp
981             ==> (k in key ++ dom rest - key)          [lemma-d]
982             ==> (k = key | k in dom rest - key) [Set.in-def]])
983     assume case1 := (k = key)
984     (!chain-> [true
985               ==> (key @ val in key @ val ++ dmap->set rest - key) [Set.in-lemma-1]
986               ==> (key @ val in dmap->set map)          [dmap->set-def]
987               ==> (exists v . key @ v in dmap->set map)  [existence]
988               ==> (exists v . k @ v in dmap->set map)    [case1]))
989     assume case2 := (k in dom rest - key)
990     (!chain-> [case2
991               ==> (exists v . k @ v in dmap->set rest - key) [IH]
992               ==> (exists v . k @ v in key @ val ++ dmap->set rest - key) [Set.in-lemma-3]
993               ==> (exists v . k @ v in dmap->set map)    [dmap->set-def]]))
994   })
995
996 conclude at-characterization-1 :=
997 (forall m k v . m at k = v ==> k @ v in dmap->set m | ~ k in dom m & v = default m)
998 pick-any m:(DMap 'K 'V) k:'K v:'V
999   assume hyp := (m at k = v)
1000   (!two-cases
1001     assume case1 := (k in dom m)
1002     pick-witness val for (!chain-> [(k in dom m)
1003                                     ==> (exists v . k @ v in dmap->set m) [ms-theorem-4]])
1004     # we now have (k @ val in dmap->set m)
1005     let {v:=val := (!chain-> [(k @ val in dmap->set m)
1006                               ==> (m at k = val)          [ms-theorem-1]
1007                               ==> (v = val)                [hyp]])}
1008     (!chain-> [(k @ val in dmap->set m)
1009               ==> (k @ v in dmap->set m)          [v:=val]
1010               ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]])
1011     assume case2 := (~ k in dom m)
1012     (!chain-> [case2
1013               ==> (m at k = default m)          [ms-theorem-2]
1014               ==> (v = default m)                [hyp]
1015               ==> (~ k in dom m & v = default m) [augment]
1016               ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]]))
1017
1018 conclude at-characterization-2 :=
1019 (forall m k v . k @ v in dmap->set m | ~ k in dom m & v = default m ==> m at k = v)
1020 pick-any m:(DMap 'K 'V) k:'K v:'V
1021   assume hyp := (k @ v in dmap->set m | ~ k in dom m & v = default m)
1022   (!cases hyp
1023     assume case1 := (k @ v in dmap->set m)
1024     (!chain-> [case1 ==> (m at k = v)          [ms-theorem-1]])
1025     assume case2 := (~ k in dom m & v = default m)
1026     (!chain-> [(~ k in dom m)
1027               ==> (m at k = default m)      [ms-theorem-2]
1028               ==> (m at k = v)              [(v = default m)])])
1029
1030 conclude at-characterization :=
1031 (forall m k v . m at k = v <==> k @ v in dmap->set m | ~ k in dom m & v = default m)
1032 pick-any m:(DMap 'K 'V) k:'K v:'V
1033   (!equiv
1034     (!chain [(m at k = v) ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [at-characterization-1]])
1035     (!chain [(k @ v in dmap->set m | ~ k in dom m & v = default m) ==> (m at k = v) [at-characterization-2]]))
1036
1037 define at-characterization-lemma :=
1038   (forall m k v . m at k = v & k in dom m ==> k @ v in dmap->set m)
1039
1040 define at-characterization-lemma-2 :=
1041   (forall m k v . m at k = v & v != default m ==> k @ v in dmap->set m)
1042
1043 (!force at-characterization-lemma)
1044 (!force at-characterization-lemma-2)
1045
1046 }
1047

```

```

1048 EOF
1049 (load "lib/basic/dmaps")
1050
1051
1052 (load "c:\\np\\book\\fsetText")
1053
1054
1055 open Pair
1056
1057 module DMap {
1058
1059 define [null ++ in subset proper-subset \/ /\ \ card A B C] :=
1060   [FSet.null FSet.++ FSet.in FSet.subset FSet.proper-subset
1061     FSet.\ FSet./\ FSet.\ FSet.card
1062     ?A:(FSet.Set 'S1) ?B:(FSet.Set 'S2) ?C:(FSet.Set 'S3)]
1063
1064 structure (DMap S T) := (empty-map T) | (update (Pair S T) (DMap S T))
1065
1066 set-precedence empty-map 250
1067
1068 define (alist->dmap-general L preprocessor) :=
1069   match L {
1070     [d (some-list pairs)] =>
1071       letrec {loop := lambda (L)
1072         match L {
1073           [] => (empty-map d)
1074           | (list-of (|| [x --> n] [x n]) rest) =>
1075             (update (pair (preprocessor x) (preprocessor n)) (loop rest))}
1076         (loop pairs)
1077     | _ => L
1078   }
1079
1080 define (alist->dmap L) := (alist->dmap-general L id)
1081
1082 define (dmap->alist-general m preprocessor) :=
1083   letrec {loop := lambda (m pairs)
1084     match m {
1085       (empty-map d) => [d (rev pairs)]
1086       | (update (pair k v) rest) =>
1087         (loop rest (add [(preprocessor k) --> (preprocessor v)] pairs))
1088       | _ => m}}
1089     (loop m [])
1090
1091 (define (remove-from m k)
1092   (match m
1093     ((empty-map _) m)
1094     ((update (binding as (pair key val)) rest)
1095       (check ((equal? k key) (remove-from rest k))
1096         (else (update binding (remove-from rest k)))))))
1097
1098 define (dmap->alist-canonical-general m preprocessor) :=
1099   letrec {loop := lambda (m pairs)
1100     match m {
1101       (empty-map d) => [d (rev pairs)]
1102       | (update (pair k v) rest) =>
1103         (loop (remove-from rest k)
1104           (add [(preprocessor k) --> (preprocessor v)] pairs))
1105       | _ => m}}
1106     (loop m [])
1107
1108 define (dmap->alist m) := (dmap->alist-general m id)
1109
1110 expand-input update [(alist->pair id id) alist->dmap]
1111
1112 declare apply: (K, V) [(DMap K V) K] -> V [110 [alist->dmap id]]
1113
1114 define [at] := [apply]
1115
1116 overload ++ update
1117

```



```

1118 set-precedence ++ 210
1119
1120 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]
1121 define [h t] := [FSet.h FSet.t]
1122
1123 define [m m' m1 m2 rest] := [?m:(DMap 'S1 'S2) ?m':(DMap 'S3 'S4) ?m1:(DMap 'S5 'S6) ?m2:(DMap 'S7 'S8) ?rest:(DMap 'S9 'S10)]
1124
1125 assert* apply-def :=
1126   [(empty-map d at _ = d)
1127    (k @ v ++ rest at x = v <== k = x)
1128    (k @ v ++ rest at x = rest at x <== k /= x)]
1129
1130 ## Some testing:
1131
1132 define make-map :=
1133   lambda (L)
1134     match L {
1135       [] => (empty-map 0)
1136       | (list-of [x n] rest) => (update (x @ n) (make-map rest))
1137     }
1138
1139 define update* :=
1140   lambda (fm pairs)
1141     letrec {loop := lambda (pairs res)
1142              match pairs {
1143                [] => res
1144                | (list-of [key val] more) => (loop more (update res key val))}}
1145     (loop pairs fm)
1146
1147
1148 define f := lambda (i) [(string->id ("s" joined-with (val->string i))) i]
1149
1150 define L := (from-to 1 5)
1151
1152 define sample-map := (make-map (map f L))
1153
1154 # So sample-map maps 's1 to 1, ..., 's5 to 5.
1155
1156 define applied-to := apply
1157
1158 (eval sample-map at 's1)
1159 (eval sample-map at 's2)
1160 (eval sample-map at 's3)
1161 (eval sample-map at 's4)
1162 (eval sample-map at 's5)
1163
1164 # And this should give the default value 0:
1165
1166 (eval sample-map at 's99)
1167
1168 declare default: (K, V) [(DMap K V)] -> V [200 [alist->dmap]]
1169
1170 assert* default-def :=
1171   [(default empty-map d = d)
1172    (default _ ++ rest = default rest)]
1173
1174 (eval default sample-map)
1175
1176 declare remove: (S, T) [(DMap S T) S] -> (DMap S T) [- 120 [alist->dmap id]]
1177
1178 left-assoc -
1179
1180 assert* remove-def :=
1181   [(empty-map d - _ = empty-map d)
1182    ([key _] ++ rest - key = rest - key)
1183    (key /= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]
1184
1185 declare dom: (S, T) [(DMap S T)] -> (FSet.Set S) [[alist->dmap]]
1186
1187 assert* dom-def :=

```

```

1188   [(dom empty-map _ = null)
1189    (dom [k v] ++ rest = dom rest - k <== v = default rest)
1190    (dom [k v] ++ rest = k ++ dom rest <== v /= default rest)]
1191
1192 declare size: (S, T) [(DMap S T)] -> N [[alist->dmap]]
1193 assert* size-axioms := [(size m = card dom m)]
1194
1195 define rc1 := (forall m x . (m - x) at x = default m)
1196
1197 by-induction rc1 {
1198   (m as (empty-map d)) =>
1199     pick-any x
1200     (!chain [(m - x at x)
1201              = (m at x)           [remove-def]
1202              = d                   [apply-def]
1203              = (default m)       [default-def]])
1204 | (m as (update (pair k:'S v) rest)) =>
1205   let {IH := (forall x . rest - x at x = default rest)}
1206     pick-any x:'S
1207     (!two-cases
1208       assume (k = x)
1209       (!chain [(m - x at x)
1210                = (m - k at k)   [(k = x)]
1211                = (rest - k at k) [remove-def]
1212                = (default rest) [IH]
1213                = (default m)   [default-def]
1214                ])
1215       assume (k /= x)
1216       (!chain [(m - x at x)
1217                = ((k @ v) ++ (rest - x) at x) [remove-def]
1218                = (rest - x at x)             [apply-def]
1219                = (default rest)              [IH]
1220                = (default m)                 [default-def]]))
1221   }
1222
1223 define rc2 := (forall m k x . k /= x ==> m - k at x = m at x)
1224
1225 by-induction rc2 {
1226   (m as (empty-map d:'V)) =>
1227     pick-any k:'K x:'K
1228     assume (k /= x)
1229     let {L := (m - k at x);
1230          R := (m at x)}
1231     (!chain [L
1232              = (m at x) [remove-def]])
1233 | (m as (update (pair key:'K val:'V) rest:(DMap 'K 'V))) =>
1234   pick-any k:'K x:'K
1235   assume (k /= x)
1236   let {IH := (forall k x . k /= x ==> (rest - k) at x = rest at x)}
1237     (!two-cases
1238       assume (key = k)
1239       let {_ := (!by-contradiction (key /= x)
1240                                   (!chain [(key = x)
1241                                              ==> (k = x)           [(key = k)]
1242                                              ==> (k = x & k /= x) [augment]
1243                                              ==> false           [prop-taut]]))}
1244       (!chain [(m - k at x)
1245                = (((k @ val) ++ rest) - k at x) [(key = k)]
1246                = (rest - k at x) [remove-def]
1247                = (rest at x)   [IH]
1248                = (m at x)     [apply-def]])
1249       assume (key /= k)
1250       (!two-cases
1251         assume (x = key)
1252         (!chain [(m - k at x)
1253                  = (((key val) ++ (rest - k)) at x) [remove-def]
1254                  = (([x val] ++ (rest - k)) at x) [(x = key)]
1255                  = val [apply-def]
1256                  = ((([x val] ++ rest) at x) [apply-def]
1257                  = (m at x) [(x = key)]))

```

```

1258         assume (x  $\neq$  key)
1259         (!chain [(m - k at x)
1260                 = (([key val] ++ (rest - k)) at x) [remove-def]
1261                 = (rest - k at x) [apply-def]
1262                 = (rest at x) [IH]
1263                 = (m at x) [apply-def]]]))
1264     }
1265
1266 define rc3 := (forall m k . default m = default m - k)
1267 by-induction rc3 {
1268   (m as (empty-map d:'V)) =>
1269     pick-any k
1270     (!chain [(default m)
1271             = (default m - k) [remove-def]])
1272 | (m as (update (pair key:'K val:'V) rest)) =>
1273   let {IH := (forall k . default rest = default rest - k)}
1274     pick-any k:'K
1275     (!two-cases
1276       assume (key = k)
1277       (!combine-equations
1278         (!chain [(default m)
1279                 = (default rest) [default-def]
1280                 = (default rest - k) [IH]])
1281         (!chain [(default m - k)
1282                 = (default rest - k) [remove-def]]))
1283       assume (key  $\neq$  k)
1284       (!chain-> [(default m - k)
1285                = (default key @ val ++ rest - k) [remove-def]
1286                = (default rest - k) [default-def]
1287                = (default rest) [IH]
1288                = (default m) [default-def]
1289                ==> (default m - k = default m)
1290                ==> (default m = default m - k) [sym]]))
1291   }
1292
1293 conclude dom-lemma-1 :=
1294   (forall k v rest . v  $\neq$  default rest ==> k in dom [k v] ++ rest)
1295 pick-any k v rest
1296   assume hyp := (v  $\neq$  default rest)
1297   (!chain-> [true ==> (k in k ++ dom rest) [FSet.in-lemma-1]
1298            ==> (k in dom [k v] ++ rest) [dom-def]])
1299
1300 conclude dom-lemma-2 :=
1301   (forall m k v . v  $\neq$  default m ==> dom m subset dom [k v] ++ m)
1302 pick-any m k v
1303   assume hyp := (v  $\neq$  default m)
1304   (!FSet.subset-intro
1305     pick-any x
1306     (!chain [(x in dom m)
1307             ==> (x in k ++ dom m) [FSet.in-lemma-3]
1308             ==> (x in dom [k v] ++ m) [dom-def]]))
1309
1310 conclude dom-lemma-2b :=
1311   (forall m x k v . v  $\neq$  default m & x in dom m ==> x in dom [k v] ++ m)
1312 pick-any m x k v
1313   assume (v  $\neq$  default m & x in dom m)
1314   let {_ := (!chain-> [(v  $\neq$  default m) ==> (dom m subset dom [k v] ++ m) [dom-lemma-2]])}
1315     (!chain-> [(x in dom m) ==> (x in dom [k v] ++ m) [FSet.SC]])
1316
1317 # conclude dom-lemma-2c :=
1318 #   (forall m x k v . x in dom [k v] ++ m ==> x = k | x in dom m - k)
1319 # pick-any m:(DMap 'K 'V) x:'K k:'K v:'V
1320 #   assume hyp := (x in dom [k v] ++ m)
1321 #   (!two-cases
1322 #     assume (v = default m)
1323 #     (!chain-> [hyp
1324                ==> (x in dom m - k) [dom-def]
1325                ==> (x = k | x in dom m - k) [prop-taut]])
1326 #     assume (v  $\neq$  default m)
1327 #     (!chain-> [hyp

```

```

1328 #           ==> (x in k ++ dom m)           [dom-def]
1329 #           ==> (x = k | x in dom m - k)     [FSet.in-def]])
1330
1331 define [< <=] := [N.< N.<=]
1332 declare len: (S, T) [(DMap S T)] -> N [[alist->dmap]]
1333
1334 assert* len-def :=
1335   [(len empty-map _ = zero)
1336    (len _ @ _ ++ rest = S len rest)]
1337
1338 define len-lemma-1 :=
1339   (forall m k v . len m < len (k @ v) ++ m)
1340
1341 by-induction len-lemma-1 {
1342   (m as (empty-map d:'V)) =>
1343     pick-any k v
1344     let {len-left := (!chain [(len m) = zero           [len-def]]);
1345          len-right := (!chain [(len k @ v ++ m) = (S len m) [len-def]]);
1346          (!chain-> [true
1347                    ==> (zero < S len m)           [N.Less.<-def]
1348                    ==> (len m < len k @ v ++ m) [len-left len-right])}
1349   | (m as (update (pair key:'K val:'V) rest)) =>
1350     let {IH := (forall k v . len rest < len k @ v ++ rest)}
1351         pick-any k:'K v:'V
1352         let {len-left := (!chain [(len m)
1353                                   = (S len rest) [len-def]]);
1354              len-right := (!chain [(len k @ v ++ m)
1355                                    = (S len m) [len-def]
1356                                    = (S S len rest) [len-left]])}
1357         (!chain-> [true
1358                   ==> (S len rest < S S len rest) [N.Less.<S]
1359                   ==> (len m < len k @ v ++ m) [len-left len-right])}
1360   }
1361
1362 conclude len-lemma-2 := (forall m k . len m - k <= len m)
1363 by-induction len-lemma-2 {
1364   (m as (empty-map d:'V)) =>
1365     pick-any k
1366     (!chain-> [(len m - k)
1367               = (len m) [remove-def]
1368               ==> (len m - k <= len m) [N.Less=.<=-def]])
1369   | (m as (update (pair key:'K val:'V) rest)) =>
1370     pick-any k:'K
1371     let {IH := (forall k . len rest - k <= len rest);
1372          L2 := (!chain-> [true ==> (len rest - k <= len rest) [IH]]);
1373          L3 := (!chain-> [true ==> (len rest < len m) [len-lemma-1]]);
1374          L4 := (!chain-> [L2 ==> (L2 & L3) [augment]
1375                          ==> (len rest - k < len m) [N.Less=.transitive2]])}
1376     (!two-cases
1377       assume (key = k)
1378         (!chain-> [(len m - k)
1379                   = (len rest - k) [remove-def]
1380                   ==> (len m - k <= len rest - k) [N.Less=.<=-def]
1381                   ==> (len m - k <= len rest - k & L2) [augment]
1382                   ==> (len m - k <= len rest) [N.Less=.transitive]
1383                   ==> (len m - k <= len rest & L3) [augment]
1384                   ==> (len m - k < len m) [N.Less=.transitive2]
1385                   ==> (len m - k <= len m) [N.Less=.<=-def]])
1386       assume (key != k)
1387         let {L5 := (!chain-> [(len m - k)
1388                               = (len [key val] ++ (rest - k)) [remove-def]
1389                               = (S len rest - k) [len-def]])}
1390         (!chain-> [L4
1391                   ==> (S len rest - k <= len m) [N.Less=.discrete]
1392                   ==> (len m - k <= len m) [L5]])}
1393   }
1394 }
1395
1396 define len-lemma-3 :=
1397   (forall key val k rest . len rest - k < len key @ val ++ rest)

```

```

1398
1399 conclude len-lemma-3
1400 pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
1401   let {m := (key @ val ++ rest);
1402       L := (!chain-> [true
1403                   ==> (len rest - k <= len rest) [len-lemma-2]])}
1404   (!chain-> [true
1405             ==> (len rest < len m)           [len-lemma-1]
1406             ==> (L & len rest < len m)       [augment]
1407             ==> (len rest - k < len m)       [N.Less=.transitive2]])
1408
1409 transform-output eval [nat->int]
1410
1411 define (lemma-D-property m) :=
1412   (forall k . k in dom m <=> m at k /= default m)
1413
1414 define lemma-D := (forall m k . k in dom m <=> m at k /= default m)
1415
1416 define lemma-D :=
1417   (forall m . lemma-D-property m)
1418
1419 (!strong-induction.measure-induction lemma-D len
1420 pick-any m:(DMap 'K 'V)
1421 assume IH := (forall m' . len m' < len m ==> lemma-D-property m')
1422 conclude (lemma-D-property m)
1423   datatype-cases (lemma-D-property m) on m {
1424     (em as (empty-map d:'V)) =>
1425       pick-any k
1426         (!equiv
1427           (!chain [(k in dom em)
1428                   ==> (k in null)      [dom-def]
1429                   ==> false            [FSet.NC]
1430                   ==> (em at k /= default em) [prop-taut]])
1431           assume h := (em at k /= default em)
1432           (!by-contradiction (k in dom em)
1433             assume (~ k in dom em)
1434               (!absurd (!reflex (default em))
1435                 (!chain-> [h ==> (d /= default em) [apply-def]
1436                           ==> (default em /= default em) [default-def]])))
1437 | (map as (update (pair key:'K val:'V) rest)) =>
1438   pick-any k:'K
1439     let {lemma1 := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
1440                             ==> (len rest - key < len m)   [(m = map)]]);
1441         lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
1442                             ==> (len rest < len m)   [(m = map)]])}
1443     (!equiv
1444       assume hyp := (k in dom map)
1445       (!two-cases
1446         assume (val = default rest)
1447         let {L1 := (!by-contradiction (k /= key)
1448                 assume (k = key)
1449                 (!absurd
1450                   (!chain [(rest - key at key)
1451                             = (default rest) [rc1]
1452                             = (default rest - key) [rc3]])
1453                   (!chain-> [(k in dom map)
1454                             ==> (key in dom map)           [(k = key)]
1455                             ==> (key in dom rest - key) [dom-def]
1456                             ==> (rest - key at key /= default rest - key) [IH]]))};
1457         _ := (!ineq-sym L1)}
1458       (!chain-> [(k in dom map)
1459               ==> (k in dom rest - key) [dom-def]
1460               ==> (rest - key at k /= default rest - key) [IH]
1461               ==> (rest - key at k /= default rest) [rc3]
1462               ==> (rest - key at k /= default map) [default-def]
1463               ==> (rest at k /= default map) [rc2]
1464               ==> (map at k /= default map) [apply-def]])
1465       assume case2 := (val /= default rest)
1466       let {M := method ()
1467           (!chain-> [(map at k) = (map at key) [(k = key)]

```

```

1468             = val [apply-def]
1469             ==> (map at k /= default rest) [case2]
1470             ==> (map at k /= default map) [default-def]]})
1471     (!cases (!chain-> [hyp
1472             ==> (k in key ++ dom rest) [dom-def]
1473             ==> (k = key | k in dom rest) [FSet.in-def]])
1474     assume (k = key)
1475     (!M)
1476     assume (k in dom rest)
1477     (!two-cases
1478     assume (k = key)
1479     (!M)
1480     assume (k /= key)
1481     (!chain-> [(k in dom rest)
1482             ==> (rest at k /= default rest) [IH]
1483             ==> (map at k /= default rest) [apply-def]
1484             ==> (map at k /= default map) [default-def]]]))))
1485 assume hyp := (map at k /= default map)
1486 (!two-cases
1487 assume case1 := (val = default rest)
1488 let {k/=key := (!by-contradiction (k /= key)
1489     assume (k = key)
1490     let {p := (!chain [(map at k)
1491             = (map at key) [(k = key)]
1492             = val [apply-def]
1493             = (default rest) [case1]
1494             = (default map) [default-def]]]}
1495     (!absurd p hyp))}
1496 (!chain-> [hyp
1497     ==> (rest at k /= default map) [apply-def]
1498     ==> ((rest - key) at k /= default map) [rc2]
1499     ==> ((rest - key) at k /= default rest) [default-def]
1500     ==> ((rest - key) at k /= default rest - key) [rc3]
1501     ==> (k in dom rest - key) [IH]
1502     ==> (k in dom map) [dom-def]])
1503 assume case2 := (val /= default rest)
1504 (!two-cases
1505 assume (k = key)
1506 (!chain<- [(k in dom map)
1507     <== (key in dom map) [(k = key)]
1508     <== (key in key ++ dom rest) [dom-def]
1509     <== true [FSet.in-lemma-1]])
1510 assume (k /= key)
1511 (!chain-> [hyp
1512     ==> (rest at k /= default map) [apply-def]
1513     ==> (rest at k /= default rest) [default-def]
1514     ==> (k in dom rest) [IH]
1515     ==> (k = key | k in dom rest) [prop-taut]
1516     ==> (k in key ++ dom rest) [FSet.in-def]
1517     ==> (k in dom map) [dom-def]]]))
1518 )
1519 })
1520
1521 conclude rc0 := (forall m x . ~ x in dom m - x)
1522 pick-any m: (DMap 'K 'V) x:'K
1523 (!by-contradiction (~ x in dom m - x)
1524 assume hyp := (x in dom m - x)
1525 (!absurd (!chain-> [true ==> (m - x at x = default m) [rc1]])
1526 (!chain-> [hyp
1527     ==> (m - x at x /= default m - x) [lemma-D]
1528     ==> (m - x at x /= default m) [rc3]]]))))
1529
1530 conclude dom-lemma-3 := (forall m k . dom (m - k) subset dom m)
1531 pick-any m: (DMap 'K 'V) k:'K
1532 (!FSet.subset-intro
1533 pick-any x:'K
1534 assume hyp := (x in dom m - k)
1535 (!two-cases
1536 assume (x = k)
1537 let {L := (!chain-> [true ==> (m - k at k = default m) [rc1]]})

```

```

1538         (!chain-> [hyp
1539             ==> (k in dom m - k)                [(x = k)]
1540             ==> (m - k at k /= default m - k)    [lemma-D]
1541             ==> (m - k at k /= default m)        [rc3]
1542             ==> (L & m - k at k /= default m)    [augment]
1543             ==> false                            [prop-taut]
1544             ==> (x in dom m)                    [prop-taut]])
1545     assume (x /= k)
1546     (!chain-> [hyp
1547         ==> (m - k at x /= default m - k)    [lemma-D]
1548         ==> (m at x /= default m - k)        [rc2]
1549         ==> (m at x /= default m)            [rc3]
1550         ==> (x in dom m)                    [lemma-D]]))
1551
1552 conclude dom-corrolary-1 :=
1553   (forall key val k rest . k in dom rest - key ==> k in dom [key val] ++ rest)
1554 pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
1555   let {L1 := (!chain-> [true ==> (dom rest - key subset dom rest)          [dom-lemma-3]])}
1556     (!two-cases
1557         assume (val = default rest)
1558         (!chain [(k in dom rest - key)
1559             ==> (k in dom [key val] ++ rest) [dom-def]])
1560         assume (val /= default rest)
1561         (!chain [(k in dom rest - key)
1562             ==> (k in dom rest)                [FSet.SC]
1563             ==> (k = key | k in dom rest)      [prop-taut]
1564             ==> (k in key ++ dom rest)        [FSet.in-def]
1565             ==> (k in dom [key val] ++ rest) [dom-def]))))
1566
1567 declare dmap->set: (K, V) [(DMap K V)] -> (FSet.Set (Pair K V)) [[alist->dmap]]
1568
1569 assert* dmap->set-def :=
1570   [(dmap->set empty-map _ = null)
1571     (dmap->set k @ v ++ rest = dmap->set rest - k <== v = default rest)
1572     (dmap->set k @ v ++ rest = (k @ v) ++ dmap->set rest - k <== v /= default rest)]
1573
1574 define ms-lemma-1a :=
1575   pick-any x key val rest v
1576     assume hyp := (x /= key)
1577     (!chain [(key _) ++ rest at x = v)
1578         <==> (rest at x = v)                [apply-def]])
1579
1580 (define (ms-lemma-1-property m)
1581   (forall k v . k @ v in dmap->set m ==> k in dom m))
1582
1583 (define ms-lemma-1
1584   (forall m (ms-lemma-1-property m)))
1585
1586 (!strong-induction.measure-induction ms-lemma-1 len
1587 pick-any m:(DMap 'K 'V)
1588   assume IH := (forall m' . len m' < len m ==> ms-lemma-1-property m')
1589   conclude (ms-lemma-1-property m)
1590     datatype-cases (ms-lemma-1-property m) on m {
1591         (em as (empty-map d:'V)) =>
1592           pick-any k v:'V
1593             (!chain [(k @ v in dmap->set em)
1594                 ==> (k @ v in FSet.null)        [dmap->set-def]
1595                 ==> false                        [FSet.NC]
1596                 ==> (k in dom em)                [prop-taut]])
1597         | (map as (update (pair key:'K val:'V) rest)) =>
1598           pick-any k:'K v:'V
1599             let {goal := (k @ v in dmap->set map ==> k in dom map);
1600                 lemma1 := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
1601                     ==> (len rest - key < len m) [(m = map)]]);
1602                 lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
1603                     ==> (len rest < len m) [(m = map)]])}
1604             (!two-cases
1605                 assume C1 := (val = default rest)
1606                 (!chain [(k @ v in dmap->set map)
1607                     ==> (k @ v in dmap->set rest - key) [dmap->set-def]

```

```

1608             ==> (k in dom rest - key) [IH]
1609             ==> (k in dom map) [dom-def]]
1610 assume C2 := (val != default rest)
1611 let {_ := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]])}
1612     (!chain [(k @ v in dmap->set map)
1613             ==> (k @ v in key @ val ++ dmap->set rest - key) [dmap->set-def]
1614             ==> (k @ v = key @ val | k @ v in dmap->set rest - key) [FSet.in-def]
1615             ==> (k = key & v = val | k @ v in dmap->set rest - key) [pair-axioms]
1616             ==> (k = key | k @ v in dmap->set rest - key) [prop-taut]
1617             ==> (k = key | k in dom rest - key) [IH]
1618             ==> (k = key | k in dom rest) [FSet.SC]
1619             ==> (k in key ++ dom rest) [FSet.in-def]
1620             ==> (k in dom map) [dom-def]])
1621     )
1622   })
1623
1624 # conclude dom-corrolary-1 :=
1625 # (forall key val k rest . k in dom rest - key ==> k in dom [key val] ++ rest)
1626 # pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
1627 # let {L1 := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]])}
1628 # (!two-cases
1629 #   assume (val = default rest)
1630 #     (!chain [(k in dom rest - key)
1631 #       ==> (k in dom [key val] ++ rest) [dom-def]])
1632 #   assume (val != default rest)
1633 #     (!chain [(k in dom rest - key)
1634 #       ==> (k in dom rest) [FSet.SC]
1635 #       ==> (k = key | k in dom rest) [prop-taut]
1636 #       ==> (k in key ++ dom rest) [FSet.in-def]
1637 #       ==> (k in dom [key val] ++ rest) [dom-def])])
1638
1639 assert* dmap-identity :=
1640 (forall m1 m2 . m1 = m2 <==> default m1 = default m2 & dmap->set m1 = dmap->set m2)
1641
1642 define dmap-identity-characterization :=
1643 (forall m1 m2 . m1 = m2 <==> forall k . m1 at k = m2 at k)
1644
1645 declare agree-on: (S, T) [(DMap S T) (DMap S T) (FSet.Set S)] -> Boolean
1646 [[alist->dmap alist->dmap FSet.lst->set]]
1647
1648
1649 assert* agree-on-def :=
1650 [(agree-on m1 m2 null)
1651  (agree-on m1 m2 h FSet.++ t) <==> m1 at h = m2 at h & (agree-on m1 m2 t))]
1652
1653 let {m1 := [77 [['x --> 1] ['y --> 2]]};
1654     m2 := [78 [['y --> 2] ['x --> 1]]};
1655 (eval (agree-on m1 m2 ['x 'y]))
1656
1657 define agreement-characterization :=
1658 (forall A m1 m2 . (agree-on m1 m2 A) <==> forall k . k in A ==> m1 at k = m2 at k)
1659
1660 by-induction agreement-characterization {
1661 (A as FSet.null:(FSet.Set 'K)) =>
1662 pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
1663 let {p1 := assume (agree-on m1 m2 A)
1664     pick-any k:'K
1665     (!chain [(k in A)
1666             ==> false [FSet.NC]
1667             ==> (m1 at k = m2 at k) [prop-taut]]);
1668     p2 := assume (forall k . k in A ==> m1 at k = m2 at k)
1669     (!chain-> [true ==> (agree-on m1 m2 A) [agree-on-def]])}
1670 (!equiv p1 p2)
1671 | (A as (FSet.insert h:'K t:(FSet.Set 'K))) =>
1672 let {IH := (forall m1 m2 . (agree-on m1 m2 t) <==> forall k . k in t ==> m1 at k = m2 at k)}
1673 pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
1674 let {p1 := assume hyp := (agree-on m1 m2 A)
1675     pick-any k:'K
1676     assume (k in A)
1677     (!cases (!chain-> [(k in A)

```



```

1678             ==> (k = h | k in t) [FSet.in-def]])
1679     assume (k = h)
1680     (!chain-> [hyp
1681       ==> (m1 at h = m2 at h) [agree-on-def]
1682       ==> (m1 at k = m2 at k) [(k = h)]])
1683     assume (k in t)
1684     let {P := (!chain-> [hyp
1685       ==> (agree-on m1 m2 t) [agree-on-def]
1686       ==> (forall k . k in t ==> m1 at k = m2 at k) [IH]])}
1687     (!chain-> [(k in t) ==> (m1 at k = m2 at k) [P]]);
1688 p2 := assume hyp := (forall k . k in A ==> m1 at k = m2 at k)
1689     let {L1 := (!chain-> [true
1690       ==> (h in A) [FSet.in-lemma-1]
1691       ==> (m1 at h = m2 at h) [hyp]])}
1692     L2 := pick-any k:'K
1693     (!chain [(k in t)
1694       ==> (k in A) [FSet.in-def]
1695       ==> (m1 at k = m2 at k) [hyp]])}
1696     L3 := (!chain-> [L2 ==> (agree-on m1 m2 t) [IH]])}
1697     (!chain-> [L1
1698       ==> (L1 & L3) [augment]
1699       ==> (agree-on m1 m2 A) [agree-on-def]])}
1700     (!equiv p1 p2)
1701 }
1702
1703 define AGC := agreement-characterization
1704
1705 conclude downward-agreement-lemma :=
1706   (forall B A m1 m2 . (agree-on m1 m2 A) & B subset A ==> (agree-on m1 m2 B))
1707 pick-any B:(FSet.Set 'K) A:(FSet.Set 'K) m1:(DMap 'K 'V) m2:(DMap 'K 'V)
1708 assume hyp := ((agree-on m1 m2 A) & B subset A)
1709 let {L := pick-any k:'K
1710   assume hyp := (k in B)
1711   (!chain-> [hyp
1712     ==> (k in A) [FSet.SC]
1713     ==> (m1 at k = m2 at k) [AGC]])}
1714     (!chain-> [L ==> (agree-on m1 m2 B) [AGC]])
1715
1716 define ms-lemma-1b := (forall m k . ~ k in dom m ==> forall v . ~ k @ v in dmap->set m)
1717
1718 by-induction ms-lemma-1b {
1719   (m as (empty-map d:'V)) =>
1720     pick-any k
1721     assume hyp := (~ k in dom m)
1722     pick-any v:'V
1723     (!by-contradiction (~ k @ v in dmap->set m)
1724       (!chain [(k @ v in dmap->set m)
1725         ==> (k @ v in FSet.null) [dmap->set-def]
1726         ==> false [FSet.NC]]))
1727 | (m as (update (pair key:'K val:'V) rest)) =>
1728   let {IH := (forall k . ~ k in dom rest ==> forall v . ~ k @ v in dmap->set rest)}
1729     pick-any k
1730     assume hyp := (~ k in dom m)
1731     pick-any v:'V
1732     (!by-contradiction (~ k @ v in dmap->set m)
1733       assume sup := (k @ v in dmap->set m)
1734       (!two-cases
1735         assume (val = default rest)
1736         (!chain-> [sup
1737           ==> (k @ v in dmap->set rest - key) [dmap->set-def]
1738           ==> (k in dom rest - key) [ms-lemma-1]
1739           ==> (k in dom m) [dom-corrolary-1]
1740           ==> (k in dom m & hyp) [augment]
1741           ==> false [prop-taut]])
1742         assume (val /= default rest)
1743         let {C :=
1744           (!chain-> [sup
1745             ==> (k @ v in key @ val FSet.++ dmap->set rest - key) [dmap->set-def]
1746             ==> (k @ v = key @ val | k @ v in dmap->set rest - key) [FSet.in-def]]);
1747           _ := (!chain-> [true ==> (dom rest - key FSet.subset dom rest) [dom-lemma-3]])}

```

```

1748     }
1749     (!cases C
1750     assume case1 := (k @ v = key @ val)
1751     let {L := (!chain-> [(val != default rest)
1752                       ==> (key in dom m) [dom-lemma-1]])}
1753     (!chain-> [case1
1754               ==> (k = key & v = val) [pair-axioms]
1755               ==> (k = key) [left-and]
1756               ==> (k in dom m) [L]
1757               ==> (k in dom m & ~ k in dom m) [augment]
1758               ==> false [prop-taut]])
1759     assume case2 := (k @ v in dmap->set rest - key)
1760     (!chain-> [case2
1761               ==> (k in dom rest - key) [ms-lemma-1]
1762               ==> (k in dom rest) [FSet.SC]
1763               ==> (k in key FSet.++ dom rest) [FSet.in-lemma-3]
1764               ==> (k in dom m) [dom-def]
1765               ==> (k in dom m & ~ k in dom m) [augment]
1766               ==> false [prop-taut]]))
1767 }
1768
1769 conclude ms-lemma-1b' := (forall m k . ~ k in dom m ==> ~ exists v . k @ v in dmap->set m)
1770 pick-any m: (DMap 'K 'V) k:'K
1771 assume h := (~ k in dom m)
1772 let {p := (!chain-> [h ==> (forall v . ~ k @ v in dmap->set m) [ms-lemma-1b']])}
1773     (!by-contradiction (~ exists v . k @ v in dmap->set m)
1774     assume hyp := (exists v . k @ v in dmap->set m)
1775     pick-witness w for hyp wp
1776     (!absurd wp (!chain-> [true ==> (~ k @ w in dmap->set m) [p]])))
1777
1778 declare restricted-to: (S, T) [(DMap S T) (FSet.Set S)] -> (DMap S T) [150 | ^ [alist->dmap FSet.lst->set]]
1779
1780 assert* restrict-axioms :=
1781     [(empty-map d | ^ _ = empty-map d)
1782     (k in A ==> [k v] ++ rest | ^ A = [k v] ++ (rest | ^ A))
1783     (~ k in A ==> [k v] ++ rest | ^ A = rest | ^ A)]
1784
1785 define sm1 := [0 [['x --> 1] ['y --> 2] ['z --> 3]]]
1786 define sm2 := [0 [['y --> 2] ['z --> 3] ['x --> 1]]]
1787
1788 (eval sm1 | ^ ['z 'y])
1789
1790 define (property m) :=
1791     (forall k v . k @ v in dmap->set m ==> m at k = v)
1792
1793 define ms-theorem-1 := (forall m . property m)
1794
1795 (!strong-induction.measure-induction ms-theorem-1 len
1796 pick-any m: (DMap 'K 'V)
1797 assume IH := (forall m' . len m' < len m ==> property m')
1798 conclude (property m)
1799 datatype-cases (property m) on m {
1800     (em as (empty-map d:'V)) =>
1801     pick-any k:'K v:'V
1802     (!chain [(k @ v in dmap->set em)
1803             ==> (k @ v in FSet.null) [dmap->set-def]
1804             ==> false [FSet.NC]
1805             ==> (em at k = v) [prop-taut]])
1806 | (map as (update (pair key:'K val:'V) rest)) =>
1807 pick-any k:'K v:'V
1808 let {goal := (k @ v in dmap->set map ==> map at k = v);
1809     lemma1 := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
1810                          ==> (len rest - key < len m) [(m = map)]];
1811     lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
1812                          ==> (len rest < len m) [(m = map)]];
1813     #lemma3 := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]]);
1814     #lemma4 := (!chain-> [true ==> (dom rest subset dom map) [dom-lemma-2]]);
1815     M := method (case)
1816         # case here must be this assumption: (k @ v in dmap->set rest - key)
1817         let {L := (!chain-> [case ==> (rest - key at k = v) [IH]]);

```

```

1818         L1 := (!chain-> [case ==> (k in dom rest - key) [ms-lemma-1]]);
1819         L2 := (!by-contradiction (k != key)
1820             assume (k = key)
1821                 (!absurd (!chain-> [true ==> (~ key in dom rest - key) [rc0]
1822                                     ==> (~ k in dom rest - key) [(k = key)]])
1823                     L1));
1824         _ := (!ineq-sym L2)
1825     (!chain-> [(key != k)
1826               ==> (rest - key at k = rest at k) [rc2]
1827               ==> (v = rest at k) [L]
1828               ==> (rest at k = v) [sym]
1829               ==> (map at k = v) [apply-def]]])
1830 (!two-cases
1831 assume (val = default rest)
1832 assume hyp := (k @ v in dmap->set map)
1833     let {L := (!chain-> [hyp ==> (k @ v in dmap->set rest - key) [dmap->set-def]]})
1834         (!M L)
1835 assume (val != default rest)
1836 assume (k @ v in dmap->set map)
1837     let {D := (!chain-> [(k @ v in dmap->set map)
1838                       ==> (k @ v in (key @ val) ++ dmap->set (rest - key)) [dmap->set-def]
1839                       ==> (k @ v = key @ val | k @ v in dmap->set (rest - key)) [FSet.in-def]]])
1840         (!cases D
1841             assume case1 := (k @ v = key @ val)
1842             let {
1843                 L1 := (!chain-> [case1
1844                                 ==> (k = key & v = val) [pair-axioms]]);
1845                 L2 := (!chain-> [(k = key) ==> (key = k) [sym]]);
1846                 L3 := (!chain-> [(v = val) ==> (val = v) [sym]]);
1847             }
1848             (!chain-> [(key = k)
1849                       ==> (map at k = val) [apply-def]
1850                       ==> (map at k = v) [(val = v)]])
1851             assume case2 := (k @ v in dmap->set (rest - key))
1852             (!M case2)))
1853     })
1854 })
1855
1856 conclude ms-theorem-2 :=
1857     (forall m k . ~ k in dom m ==> m at k = default m)
1858 pick-any m: (DMap 'K 'V) k:'K
1859     assume hyp := (~ k in dom m)
1860     (!chain-> [hyp ==> (~ m at k != default m) [lemma-D]
1861               ==> (m at k = default m) [dn]])
1862
1863 define lemma-q := (forall m k k' . k in dom m & k != k' ==> k in dom m - k')
1864
1865 by-induction lemma-q {
1866     (m as (empty-map d:'V)) =>
1867     pick-any k k'
1868         assume hyp := (k in dom m & k != k')
1869         (!chain-> [(k in dom m)
1870                   ==> (k in FSet.null) [dom-def]
1871                   ==> false [FSet.NC]
1872                   ==> (k in dom m - k') [prop-taut]])
1873 | (m as (update (pair key:'K val:'V) rest)) =>
1874     pick-any k:'K k':'K
1875         assume hyp := (k in dom m & k != k')
1876         (!two-cases
1877             assume (val = default rest)
1878             let {
1879                 _ := (!chain-> [true ==> (dom rest - key subset dom rest) [dom-lemma-3]]);
1880                 case2 := (!chain-> [(k in dom m)
1881                                     ==> (k in dom rest - key) [dom-def]
1882                                     ==> (k in dom rest) [FSet.SC]]);
1883                 IH := (forall k k' . k in dom rest & k != k' ==> k in dom rest - k');
1884                 L := (!chain-> [case2
1885                               ==> (case2 & k != k') [augment]
1886                               ==> (k in dom rest - k') [IH]])
1887             }

```

```

1888 (!two-cases
1889   assume (key = k')
1890   (!chain-> [L
1891     ==> (k in dom rest - key) [(key = k')]
1892     ==> (k in dom m - key) [remove-def]
1893     ==> (k in dom m - k') [(key = k')]])
1894   assume (key /= k')
1895   let {_ := ()};
1896   p := (!chain [(dom (key @ val) ++ (rest - k'))
1897     = (key ++ dom rest - k') [dom-def]])
1898   }
1899   (!chain-> [L
1900     ==> (k in key ++ dom rest - k') [FSet.in-lemma-3]
1901     ==> (k in dom (key @ val) ++ (rest - k')) [p]
1902     ==> (k in dom m - k') [remove-def]]))
1903 assume (val /= default rest)
1904 let {C := (!chain-> [(k in dom m)
1905   ==> (k in key ++ dom rest) [dom-def]
1906   ==> (k = key | k in dom rest) [FSet.in-def]])}
1907 (!cases C
1908   assume case1 := (k = key)
1909   let {_ := ()};
1910   _ := (!chain-> [(k /= k')
1911     ==> (key /= k') [case1]]);
1912   _ := (!claim (val /= default rest));
1913   L := (!chain [(dom (key @ val) ++ (rest - k'))
1914     = (key ++ dom (rest - k')) [dom-def]]);
1915   ## BUG: YOU SHOULDN'T HAVE TO FORMULATE L separately here.
1916   ## It should be a normal part of the following chain:
1917   _ := ()
1918   }
1919   (!chain-> [true
1920     ==> (key in key ++ dom rest - k') [FSet.in-lemma-1]
1921     ==> (k in key ++ dom (rest - k')) [(k = key)]
1922     ==> (k in dom (key @ val) ++ (rest - k')) [L]
1923     ==> (k in dom m - k') [remove-def]])
1924   assume case2 := (k in dom rest)
1925   let {IH := (forall k k' . k in dom rest & k /= k' ==> k in dom rest - k');
1926     L := (!chain-> [case2
1927       ==> (case2 & k /= k') [augment]
1928       ==> (k in dom rest - k') [IH]]
1929     )}
1930   (!two-cases
1931     assume (key = k')
1932     (!chain-> [L
1933       ==> (k in dom rest - key) [(key = k')]
1934       ==> (k in dom m - key) [remove-def]
1935       ==> (k in dom m - k') [(key = k')]])
1936     assume (key /= k')
1937     let {_ := ()};
1938     p := (!chain [(dom (key @ val) ++ (rest - k'))
1939       = (key ++ dom rest - k') [dom-def]]);
1940     # SAME PROBLEM WITH P HERE. SHOULDN'T HAVE TO DO IT
1941     # SEPARATELY BY ITSELF TO USE IT IN THE CHAIN BELOW.
1942     # I SHOULD BE ABLE TO SAY [DOM-DEF] IN THE STEP BELOW
1943     # (RATHER THAN [P]).
1944     _ := ()
1945     }
1946     (!chain-> [L
1947       ==> (k in key ++ dom rest - k') [FSet.in-lemma-3]
1948       ==> (k in dom (key @ val) ++ (rest - k')) [p]
1949       ==> (k in dom m - k') [remove-def]]))
1950   }
1951
1952 conclude lemma-d :=
1953   (forall m key val . val /= default m ==> dom key @ val ++ m = key ++ dom m - key)
1954 pick-any m:(DMap 'K 'V) key:'K val:'V
1955   assume (val /= default m)
1956   let {L := (dom key @ val ++ m);
1957     R := (key ++ dom m - key);

```

```

1958 R->L := (!FSet.subset-intro
1959   pick-any k:'K
1960   assume (k in R)
1961     (!cases (!chain-> [(k in R)
1962       ==> (k = key | k in dom m - key) [FSet.in-def]])
1963     assume (k = key)
1964     (!chain-> [true
1965       ==> (key in key ++ dom m) [FSet.in-lemma-1]
1966       ==> (key in dom key @ val ++ m) [dom-def]
1967       ==> (k in L) [(k = key)]])
1968     assume case2 := (k in dom m - key)
1969     let {_ := (!chain-> [true ==> (dom m - key subset dom m) [dom-lemma-3]])}
1970     (!chain-> [case2
1971       ==> (k in dom m) [FSet.SC]
1972       ==> (k in key ++ dom m) [FSet.in-lemma-3]
1973       ==> (k in L) [dom-def]]));
1974 L->R := (!FSet.subset-intro
1975   pick-any k:'K
1976   assume (k in L)
1977   let {M := method ()
1978     (!chain-> [true
1979       ==> (key in key ++ dom m - key) [FSet.in-lemma-1]
1980       ==> (k in R) [(k = key)]])
1981     (!cases (!chain-> [(k in L)
1982       ==> (k in key ++ dom m) [dom-def]
1983       ==> (k = key | k in dom m) [FSet.in-def]])
1984     assume (k = key)
1985     (!M)
1986     assume (k in dom m)
1987     (!two-cases
1988       assume (k = key)
1989       (!M)
1990       assume (k /= key)
1991       (!chain-> [(k in dom m)
1992         ==> (k in dom m & k /= key) [augment]
1993         ==> (k in dom m - key) [lemma-q]
1994         ==> (k in R) [FSet.in-def]]))
1995     (!FSet.set-identity-intro L->R R->L))
1996
1997 define (ms-theorem-4-property m) :=
1998   (forall k . k in dom m ==> exists v . k @ v in dmap->set m)
1999
2000 define ms-theorem-4 := (forall m . ms-theorem-4-property m)
2001
2002 (!strong-induction.measure-induction ms-theorem-4 len
2003   pick-any m:(DMap 'K 'V)
2004   assume IH := (forall m' . len m' < len m ==> ms-theorem-4-property m')
2005   conclude (ms-theorem-4-property m)
2006   datatype-cases (ms-theorem-4-property m) on m {
2007     (em as (empty-map d:'V)) =>
2008       pick-any k:'K
2009       (!chain [(k in dom em)
2010         ==> (k in FSet.null) [dom-def]
2011         ==> false [FSet.NC]
2012         ==> (exists v . k @ v in dmap->set em) [prop-taut]])
2013     | (map as (update (pair key:'K val:'V) rest)) =>
2014       pick-any k:'K
2015       let {lemmal := (!chain-> [true ==> (len rest - key < len map) [len-lemma-3]
2016         ==> (len rest - key < len m) [(m = map)]]);
2017         lemma2 := (!chain-> [true ==> (len rest < len map) [len-lemma-1]
2018         ==> (len rest < len m) [(m = map)]]);
2019         _ := ()
2020       }
2021       assume hyp := (k in dom map)
2022       (!two-cases
2023         assume (val = default rest)
2024         (!chain-> [hyp
2025           ==> (k in dom rest - key) [dom-def]
2026           ==> (exists v . k @ v in dmap->set rest - key) [IH]
2027           ==> (exists v . k @ v in dmap->set map) [dmap->set-def]]))

```

```

2028     assume (val != default rest)
2029     (!chain-> [hyp
2030         ==> (k in key ++ dom rest - key)      [lemma-d]
2031         ==> (k = key | k in dom rest - key) [FSet.in-def]])
2032     assume case1 := (k = key)
2033     (!chain-> [true
2034         ==> (key @ val in key @ val ++ dmap->set rest - key) [FSet.in-lemma-1]
2035         ==> (key @ val in dmap->set map)                       [dmap->set-def]
2036         ==> (exists v . key @ v in dmap->set map)             [existence]
2037         ==> (exists v . k @ v in dmap->set map)               [case1]])
2038     assume case2 := (k in dom rest - key)
2039     (!chain-> [case2
2040         ==> (exists v . k @ v in dmap->set rest - key) [IH]
2041         ==> (exists v . k @ v in key @ val ++ dmap->set rest - key) [FSet.in-lemma-3]
2042         ==> (exists v . k @ v in dmap->set map)           [dmap->set-def]]))
2043 })
2044
2045 conclude at-characterization-1 :=
2046 (forall m k v . m at k = v ==> k @ v in dmap->set m | ~ k in dom m & v = default m)
2047 pick-any m:(DMap 'K 'V) k:'K v:'V
2048     assume hyp := (m at k = v)
2049     (!two-cases
2050         assume case1 := (k in dom m)
2051         pick-witness val for (!chain-> [(k in dom m)
2052             ==> (exists v . k @ v in dmap->set m) [ms-theorem-4]])
2053         # we now have (k @ val in dmap->set m)
2054         let {v=val := (!chain-> [(k @ val in dmap->set m)
2055             ==> (m at k = val)                [ms-theorem-1]
2056             ==> (v = val)                      [hyp]])}
2057         (!chain-> [(k @ val in dmap->set m)
2058             ==> (k @ v in dmap->set m)          [v=val]
2059             ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]])
2060         assume case2 := (~ k in dom m)
2061         (!chain-> [case2
2062             ==> (m at k = default m)           [ms-theorem-2]
2063             ==> (v = default m)                [hyp]
2064             ==> (~ k in dom m & v = default m) [augment]
2065             ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]]))
2066
2067 conclude at-characterization-2 :=
2068 (forall m k v . k @ v in dmap->set m | ~ k in dom m & v = default m ==> m at k = v)
2069 pick-any m:(DMap 'K 'V) k:'K v:'V
2070     assume hyp := (k @ v in dmap->set m | ~ k in dom m & v = default m)
2071     (!cases hyp
2072         assume case1 := (k @ v in dmap->set m)
2073         (!chain-> [case1 ==> (m at k = v)      [ms-theorem-1]])
2074         assume case2 := (~ k in dom m & v = default m)
2075         (!chain-> [(~ k in dom m)
2076             ==> (m at k = default m) [ms-theorem-2]
2077             ==> (m at k = v)         [(v = default m)])])
2078
2079 conclude at-characterization :=
2080 (forall m k v . m at k = v <==> k @ v in dmap->set m | ~ k in dom m & v = default m)
2081 pick-any m:(DMap 'K 'V) k:'K v:'V
2082     (!equiv
2083         (!chain [(m at k = v) ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [at-characterization-1]])
2084         (!chain [(k @ v in dmap->set m | ~ k in dom m & v = default m) ==> (m at k = v) [at-characterization-2]]))
2085
2086 define at-characterization-lemma :=
2087     (forall m k v . m at k = v & k in dom m ==> k @ v in dmap->set m)
2088
2089 define at-characterization-lemma-2 :=
2090     (forall m k v . m at k = v & v != default m ==> k @ v in dmap->set m)
2091
2092 (!force at-characterization-lemma)
2093 (!force at-characterization-lemma-2)
2094
2095 } # module DMap
2096
2097 EOF

```

```

2098 (load "c:\\np\\book\\dmapText2")
2099
2100 #START_LOAD
2101
2102 define map-identity-characterization-1 :=
2103   (forall m1 m2 . (forall k . m1 at k = m2 at k) ==> m1 = m2)
2104
2105 conclude map-identity-characterization-1
2106   pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
2107     assume hyp := (forall k . m1 at k = m2 at k)
2108       let {L1 := conclude (dmap->set m1 = dmap->set m2)
2109           (!FSet.set-identity-intro-direct
2110             (!pair-converter
2111               pick-any k:'K v:'V
2112                 (!equiv
2113                   assume hyp' := (k @ v in dmap->set m1)
2114                     let {L := (!chain-> [hyp ==> (k in dom m1) [ms-lemma-1]])}
2115                         (!chain-> [(k @ v in dmap->set m1)
2116                                   ==> (m1 at k = v)           [at-characterization]
2117                                   ==> (m1 at k = v & L)       [augment]
2118                                   ==> (m2 at k = v & L)       [hyp]
2119                                   ==> (m2 at k = v)           [hyp]
2120                                   ==> (k @ v in dmap->set m2)   [at-characterization])]}
2121                             (!chain [(k @ v in dmap->set m2)
2122                                       ==> (m2 at k = v)       [at-characterization]
2123                                       ==> (m1 at k = v)       [hyp]
2124                                       ==> (k @ v in dmap->set m1) [at-characterization])])
2125
2126           ;
2127           L2 := conclude (default m1 = default m2)
2128                 (!force (default m1 = default m2))
2129           ; _ := (halt)
2130         }
2131       (!chain-> [L2
2132                 ==> (L2 & L1) [augment]
2133                 ==> (m1 = m2) [dmap-identity]])
2134 #END_LOAD
2135 (load "c:\\np\\book\\dmapText2")
2136
2137 define identity-result-1 :=
2138   (forall m1 m2 . (forall k . m1 at k = m2 at k) ==>
2139     dmap->set m1 = dmap->set m2 & default m1 = default m2)
2140
2141
2142 #START_LOAD
2143
2144 conclude identity-result-1
2145   pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
2146     assume hyp := (forall k . m1 at k = m2 at k)
2147       let {[s1 s2] := [(dmap->set m1) (dmap->set m2)];
2148           L1 :=
2149             (!by-contradiction (s1 = s2)
2150               assume hyp' := (s1 /= s2)
2151                 (!cases (!chain-> [hyp' ==> ((exists p . p in s1 & ~ p in s2) |
2152                                             (exists p . p in s2 & ~ p in s1))
2153                                       [FSet.neg-set-identity-characterization-2]])
2154                   assume case1 := (exists p . p in s1 & ~ p in s2)
2155                     let {case1 := conclude (exists k v . k @ v in s1 & ~ k @ v in s2)
2156                         (!pair-converter-2 case1)}
2157                       pick-witnesses k v for case1
2158
2159                   assume case2 := (exists p . p in s2 & ~ p in s1)
2160                     (!dhalt))}
2161             (!dhalt)
2162 #END_LOAD
2163 # (load "c:\\np\\book\\dmapText")

```