lib/basic/dmaps.ath

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

load "sets"
load "strong-induction"

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]

overload ++ update
```

```
set-precedence ++ 210

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]
define [h t] := [Set.h Set.t]

define [m m' m1 m2 rest] := [?m:(DMap 'S1 'S2) ?m':(DMap 'S3 'S4) ?m1:(DMap 'S5 'S6) ?m2:(DMap 'S7 'S8) ?rest:(DMap 'S

assert* apply-def :=
  [(empty-map d at _ = d)
   (k @ v ++ rest at x = v <== k = x)
   (k @ v ++ rest at x =  rest at x <== k =/= x)]

## Some testing:

define make-map :=
  lambda (L)
    match L {
       [] => (empty-map 0)
     | (list-of [x n] rest) => (update (x @ n) (make-map rest))
    }

define update* :=
  lambda (fm pairs)
    letrec {loop := lambda (pairs res)
                      match pairs {
                        [] => res
                      | (list-of [key val] more) => (loop more (update res key val))}}
       (loop pairs fm)


define f := lambda (i) [(string->id ("s" joined-with (val->string i))) i]

define L := (from-to 1 5)

define sample-map := (make-map (map f L))

# So sample-map maps 's1 to 1, ..., 's5 to 5.

define applied-to := apply

(eval sample-map at 's1)
(eval sample-map at 's2)
(eval sample-map at 's3)
(eval sample-map at 's4)
(eval sample-map at 's5)

# And this should give the default value 0:

(eval sample-map at 's99)

declare default: (K, V) [(DMap K V)] -> V [200 [alist->dmap]]

assert* default-def :=
  [(default empty-map d = d)
   (default _ ++ rest = default rest)]

(eval default sample-map)

declare remove: (S, T) [(DMap S T) S] -> (DMap S T) [- 120 [alist->dmap id]]

left-assoc -

assert* remove-def :=
  [(empty-map d - _ = empty-map d)
   ([key _] ++ rest - key = rest - key)
   (key =/= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]

declare dom: (S, T) [(DMap S T)] -> (Set.Set S)   [[alist->dmap]]
```

```
assert* dom-def :=
  [(dom empty-map _ = null)
   (dom [k v] ++ rest = dom rest - k <== v = default rest)
   (dom [k v] ++ rest = k ++ dom rest <== v =/= default rest)]

declare size: (S, T) [(DMap S T)] -> N [[alist->dmap]]
assert* size-axioms := [(size m = card dom m)]

define rc1 := (forall m x . (m - x) at x = default m)

by-induction rc1 {
  (m as (empty-map d)) =>
     pick-any x
      (!chain [(m - x at x)
             = (m at x)      [remove-def]
             = d             [apply-def]
             = (default m)   [default-def]])
| (m as (update (pair k:'S v) rest)) =>
    let {IH := (forall x . rest - x at x = default rest)}
      pick-any x:'S
        (!two-cases
           assume (k = x)
            (!chain [(m - x at x)
                   = (m - k at k)    [(k = x)]
                   = (rest - k at k)    [remove-def]
                   = (default rest)     [IH]
                   = (default m)        [default-def]
                    ])
           assume (k =/= x)
             (!chain [(m - x at x)
                    = ((k @ v) ++ (rest - x) at x)  [remove-def]
                    = (rest - x at x)               [apply-def]
                    = (default rest)                [IH]
                    = (default m)                   [default-def]]))
}

define rc2 := (forall m k x . k =/= x ==> m - k at x = m at x)

by-induction rc2 {
 (m as (empty-map d:'V)) =>
   pick-any k:'K x:'K
     assume (k =/= x)
       let {L := (m - k at x);
            R := (m at x)}
         (!chain [L
               = (m at x)  [remove-def]])
| (m as (update (pair key:'K val:'V) rest:(DMap 'K 'V))) =>
   pick-any k:'K x:'K
     assume (k =/= x)
       let {IH := (forall k x . k =/= x ==> (rest - k) at x = rest at x)}
         (!two-cases
           assume (key = k)
            let {_ := (!by-contradiction (key =/= x)
                        (!chain [(key = x)
                             ==> (k = x)            [(key = k)]
                             ==> (k = x & k =/= x)  [augment]
                             ==> false              [prop-taut]]))}
            (!chain [(m - k at x)
                   = (((k @ val) ++ rest) - k at x)   [(key = k)]
                   = (rest - k at x)   [remove-def]
                   = (rest at x)       [IH]
                   = (m at x)          [apply-def]])
           assume (key =/= k)
             (!two-cases
                assume (x = key)
                  (!chain [(m - k at x)
                        = (([key val] ++ (rest - k)) at x)    [remove-def]
                        = (([x val] ++ (rest - k)) at x)    [(x = key)]
                        = val                               [apply-def]
                        = (([x val] ++ rest) at x)          [apply-def]
```

```
                        = (m at x)                              [(x = key)]])
                assume (x =/= key)
                  (!chain [(m - k at x)
                        = (([key val] ++ (rest - k)) at x)    [remove-def]
                        = (rest - k at x)                     [apply-def]
                        = (rest at x)                         [IH]
                        = (m at x)                            [apply-def]])))
}

define rc3 := (forall m k . default m = default m - k)
by-induction rc3 {
  (m as (empty-map d:'V)) =>
     pick-any k
       (!chain [(default m)
              = (default m - k) [remove-def]])
 | (m as (update (pair key:'K val:'V) rest)) =>
     let {IH := (forall k . default rest = default rest - k)}
     pick-any k:'K
       (!two-cases
          assume (key = k)
           (!combine-equations
             (!chain [(default m)
                    = (default rest)      [default-def]
                    = (default rest - k)  [IH]])
             (!chain [(default m - k)
                    = (default rest - k)  [remove-def]]))
          assume (key =/= k)
            (!chain-> [(default m - k)
                     = (default key @ val ++ rest - k) [remove-def]
                     = (default rest - k)              [default-def]
                     = (default rest)                  [IH]
                     = (default m)                     [default-def]
                   ==> (default m - k = default m)
                   ==> (default m = default m - k)     [sym]]))
}

conclude dom-lemma-1 :=
  (forall k v rest . v =/= default rest ==> k in dom [k v] ++ rest)
pick-any k v rest
  assume hyp := (v =/= default rest)
  (!chain-> [true ==> (k in k ++ dom rest)     [Set.in-lemma-1]
                  ==> (k in dom [k v] ++ rest) [dom-def]])

conclude dom-lemma-2 :=
  (forall m k v . v =/= default m ==> dom m subset dom [k v] ++ m)
pick-any m k v
  assume hyp := (v =/= default m)
  (!Set.subset-intro
     pick-any x
        (!chain [(x in dom m)
             ==> (x in k ++ dom m)      [Set.in-lemma-3]
             ==> (x in dom [k v] ++ m)  [dom-def]]))

conclude dom-lemma-2b :=
  (forall m x k v . v =/= default m & x in dom m ==> x in dom [k v] ++ m)
pick-any m x k v
  assume (v =/= default m & x in dom m)
  let {_ := (!chain-> [(v =/= default m) ==> (dom m subset dom [k v] ++ m) [dom-lemma-2]])}
    (!chain-> [(x in dom m) ==> (x in dom [k v] ++ m) [Set.SC]])

# conclude dom-lemma-2c :=
#   (forall m x k v . x in dom [k v] ++ m ==> x = k | x in dom m - k)
# pick-any m:(DMap 'K 'V) x:'K k:'K v:'V
#   assume hyp := (x in dom [k v] ++ m)
#     (!two-cases
#       assume (v = default m)
#         (!chain-> [hyp
#                ==> (x in dom m - k)          [dom-def]
#                ==> (x = k | x in dom m - k) [prop-taut]])
#       assume (v =/= default m)
```

```
#          (!chain-> [hyp
#                 ==> (x in k ++ dom m)          [dom-def]
#                 ==> (x = k | x in dom m - k)       [Set.in-def]]))

define [< <=] := [N.< N.<=]
declare len: (S, T) [(DMap S T)] -> N [[alist->dmap]]

assert* len-def :=
  [(len empty-map _ = zero)
   (len _ @ _ ++ rest = S len rest)]

define len-lemma-1 :=
  (forall m k v . len m < len (k @ v) ++ m)

by-induction len-lemma-1 {
  (m as (empty-map d:'V)) =>
    pick-any k v
      let {len-left :=  (!chain [(len m) = zero                [len-def]]);
           len-right := (!chain [(len k @ v ++ m) = (S len m) [len-def]])}
       (!chain-> [true
              ==> (zero < S len m)          [N.Less.<-def]
              ==> (len m < len k @ v ++ m) [len-left len-right]])
 | (m as (update (pair key:'K val:'V) rest)) =>
     let {IH := (forall k v . len rest < len k @ v ++ rest)}
       pick-any k:'K v:'V
         let {len-left := (!chain [(len m)
                                  = (S len rest)   [len-def]]);
              len-right := (!chain [(len k @ v ++ m)
                                   = (S len m)      [len-def]
                                   = (S S len rest) [len-left]])}
          (!chain-> [true
                 ==> (S len rest < S S len rest)  [N.Less.<S]
                 ==> (len m < len k @ v ++ m)     [len-left len-right]])
}

conclude len-lemma-2 := (forall m k . len m - k <= len m)
by-induction len-lemma-2 {
  (m as (empty-map d:'V)) =>
    pick-any k
    (!chain-> [(len m - k)
             = (len m)              [remove-def]
          ==>  (len m - k <= len m) [N.Less=.<=-def]])
 | (m as (update (pair key:'K val:'V) rest)) =>
     pick-any k:'K
       let {IH := (forall k . len rest - k <= len rest);
            L2 := (!chain-> [true ==> (len rest - k <= len rest) [IH]]);
            L3 := (!chain-> [true ==> (len rest < len m)          [len-lemma-1]]);
            L4 := (!chain-> [L2 ==> (L2 & L3)                     [augment]
                               ==> (len rest - k < len m)        [N.Less=.transitive2]])}
        (!two-cases
          assume (key = k)
            (!chain-> [(len m - k)
                     = (len rest - k)                  [remove-def]
                   ==> (len m - k <= len rest - k)       [N.Less=.<=-def]
                   ==> (len m - k <= len rest - k & L2) [augment]
                   ==> (len m - k <= len rest)          [N.Less=.transitive]
                   ==> (len m - k <= len rest & L3)      [augment]
                   ==> (len m - k < len m)              [N.Less=.transitive2]
                   ==> (len m - k <= len m)             [N.Less=.<=-def]])
          assume (key =/= k)
            let {L5 := (!chain-> [(len m - k)
                               = (len [key val] ++ (rest - k)) [remove-def]
                               = (S len rest - k)              [len-def]])}

              (!chain-> [L4
                     ==> (S len rest - k <= len m)  [N.Less=.discrete]
                     ==> (len m - k <= len m)       [L5]]))
}

define len-lemma-3 :=
```

```
  (forall key val k rest . len rest - k < len key @ val ++ rest)

conclude len-lemma-3
  pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
    let {m := (key @ val ++ rest);
         L := (!chain-> [true
                     ==> (len rest - k <= len rest) [len-lemma-2]])}
      (!chain-> [true
             ==> (len rest < len m)          [len-lemma-1]
             ==> (L & len rest < len m)      [augment]
             ==> (len rest - k < len m)      [N.Less=.transitive2]])

transform-output eval [nat->int]

define (lemma-D-property m) :=
  (forall k .  k in dom m <==> m at k =/= default m)

define lemma-D := (forall m k . k in dom m <==> m at k =/= default m)

define lemma-D :=
  (forall m . lemma-D-property m)

(!strong-induction.measure-induction lemma-D len
pick-any m:(DMap 'K 'V)
  assume IH := (forall m' . len m' < len m ==> lemma-D-property m')
    conclude (lemma-D-property m)
      datatype-cases (lemma-D-property m) on m  {
        (em as (empty-map d:'V)) =>
         pick-any k
           (!equiv
             (!chain [(k in dom em)
                  ==> (k in null)   [dom-def]
                  ==> false         [Set.NC]
                  ==> (em at k =/= default em) [prop-taut]])
            assume h := (em at k =/= default em)
              (!by-contradiction (k in dom em)
                assume (~ k in dom em)
                  (!absurd (!reflex (default em))
                           (!chain-> [h ==> (d =/= default em)          [apply-def]
                                        ==> (default em =/= default em) [default-def]]))))
      | (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)]])}
             (!equiv
              assume hyp := (k in dom map)
                (!two-cases
                  assume (val = default rest)
                    let {L1 := (!by-contradiction (k =/= key)
                                 assume (k = key)
                                   (!absurd
                                    (!chain [(rest - key at key)
                                           = (default rest)  [rc1]
                                           = (default rest - key) [rc3]])
                                    (!chain-> [(k in dom map)
                                           ==> (key in dom map)        [(k = key)]
                                           ==> (key in dom rest - key) [dom-def]
                                           ==> (rest - key at key =/= default rest - key) [IH]])));
                         _ := (!ineq-sym L1)}
                     (!chain-> [(k in dom map)
                             ==> (k in dom rest - key)  [dom-def]
                             ==> (rest - key at k =/= default rest - key)  [IH]
                             ==> (rest - key at k =/= default rest)        [rc3]
                             ==> (rest - key at k =/= default map)         [default-def]
                             ==> (rest at k =/= default map)               [rc2]
                             ==> (map at k =/= default map)               [apply-def]])
                  assume case2 := (val =/= default rest)
                   let {M := method ()
```

```
                             (!chain-> [(map at k) = (map at key)      [(k = key)]
                                                   = val               [apply-def]
                                    ==> (map at k =/= default rest)    [case2]
                                    ==> (map at k =/= default map)     [default-def]])}
                   (!cases (!chain-> [hyp
                                   ==> (k in key ++ dom rest)     [dom-def]
                                   ==> (k = key | k in dom rest) [Set.in-def]])
                      assume (k = key)
                        (!M)
                      assume (k in dom rest)
                        (!two-cases
                          assume (k = key)
                            (!M)
                          assume (k =/= key)
                            (!chain-> [(k in dom rest)
                                   ==> (rest at k =/= default rest) [IH]
                                   ==> (map at k  =/= default rest) [apply-def]
                                   ==> (map at k  =/= default map)  [default-def]]))))
              assume hyp := (map at k =/= default map)
                (!two-cases
                   assume case1 := (val = default rest)
                    let {k=/=key := (!by-contradiction (k =/= key)
                                       assume (k = key)
                                        let {p := (!chain [(map at k)
                                                         = (map at key)    [(k = key)]
                                                         = val             [apply-def]
                                                         = (default rest) [case1]
                                                         = (default map)  [default-def]])}
                                        (!absurd p hyp))}
                      (!chain-> [hyp
                             ==> (rest at k =/= default map) [apply-def]
                             ==> ((rest - key) at k =/= default map) [rc2]
                             ==> ((rest - key) at k =/= default rest) [default-def]
                             ==> ((rest - key) at k =/= default rest - key) [rc3]
                             ==> (k in dom rest - key)                      [IH]
                             ==> (k in dom map)                             [dom-def]])
                   assume case2 := (val =/= default rest)
                       (!two-cases
                         assume (k = key)
                           (!chain<- [(k in dom map)
                                  <== (key in dom map) [(k = key)]
                                  <== (key in key ++ dom rest) [dom-def]
                                  <== true                     [Set.in-lemma-1]])
                         assume (k =/= key)
                           (!chain-> [hyp
                                  ==> (rest at k =/= default map)  [apply-def]
                                  ==> (rest at k =/= default rest) [default-def]
                                  ==> (k in dom rest)              [IH]
                                  ==> (k = key | k in dom rest)    [prop-taut]
                                  ==> (k in key ++ dom rest)       [Set.in-def]
                                  ==> (k in dom map)               [dom-def]])))
       )
     })

conclude rc0 := (forall m x . ~ x in dom m - x)
  pick-any m:(DMap 'K 'V)  x:'K
    (!by-contradiction (~ x in dom m - x)
      assume hyp := (x in dom m - x)
       (!absurd (!chain-> [true ==> (m - x at x = default m) [rc1]])
                (!chain-> [hyp
                       ==> (m - x at x =/= default m - x)    [lemma-D]
                       ==> (m - x at x =/= default m)        [rc3]])))

conclude dom-lemma-3 := (forall m k . dom (m - k) subset dom m)
pick-any m:(DMap 'K 'V) k:'K
(!Set.subset-intro
  pick-any x:'K
    assume hyp := (x in dom m - k)
      (!two-cases
         assume (x = k)
```

```
          let {L := (!chain-> [true ==> (m - k at k = default m) [rc1]])}
             (!chain-> [hyp
                    ==> (k in dom m - k)                [(x = k)]
                    ==> (m - k at k =/= default m - k)  [lemma-D]
                    ==> (m - k at k =/= default m)      [rc3]
                    ==> (L & m - k at k =/= default m)  [augment]
                    ==> false                           [prop-taut]
                    ==> (x in dom m)                    [prop-taut]])
         assume (x =/= k)
           (!chain-> [hyp
                  ==> (m - k at x =/= default m - k)   [lemma-D]
                  ==> (m at x =/= default m - k)       [rc2]
                  ==> (m at x =/= default m)           [rc3]
                  ==> (x in dom m)                     [lemma-D]])))

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]]))

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.lst->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))]

let {m1 := [77 [['x --> 1] ['y --> 2]]];
     m2 := [78 [['y --> 2] ['x --> 1]]]}
 (eval (agree-on m1 m2 ['x 'y]))

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)
}

define AGC := agreement-characterization

conclude downward-agreement-lemma :=
  (forall B A m1 m2 . (agree-on m1 m2 A) & B subset A ==> (agree-on m1 m2 B))
pick-any B:(Set.Set 'K) A:(Set.Set 'K) m1:(DMap 'K 'V) m2:(DMap 'K 'V)
  assume hyp := ((agree-on m1 m2 A) & B subset A)
    let {L := pick-any k:'K
                assume hyp := (k in B)
                  (!chain-> [hyp
                         ==> (k in A) [Set.SC]
                         ==> (m1 at k = m2 at k)  [AGC]])}
      (!chain-> [L ==> (agree-on m1 m2 B) [AGC]])

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

by-induction ms-lemma-1b {
  (m as (empty-map d:'V)) =>
     pick-any k
      assume hyp := (~ k in dom m)
        pick-any v:'V
          (!by-contradiction (~ k @ v in dmap->set m)
             (!chain [(k @ v in dmap->set m)
                  ==> (k @ v in Set.null)     [dmap->set-def]
                  ==> false                    [Set.NC]]))
| (m as (update (pair key:'K val:'V) rest)) =>
    let {IH :=  (forall k . ~ k in dom rest ==> forall v . ~ k @ v in dmap->set rest)}
      pick-any k
        assume hyp := (~ k in dom m)
          pick-any v:'V
            (!by-contradiction (~ k @ v in dmap->set m)
               assume sup := (k @ v in dmap->set m)
               (!two-cases
                assume (val = default rest)
                  (!chain-> [sup
                         ==> (k @ v in dmap->set rest - key)  [dmap->set-def]
                         ==> (k in dom rest - key)            [ms-lemma-1]
                         ==> (k in dom m)                     [dom-corrolary-1]
                         ==> (k in dom m & hyp)               [augment]
                         ==> false                            [prop-taut]])
                assume (val =/= default rest)
                let {C :=
                       (!chain-> [sup
                             ==> (k @ v in key @ val Set.++ dmap->set rest - key)    [dmap->set-def]
                             ==> (k @ v = key @ val | k @ v in dmap->set rest - key)  [Set.in-def]]);
```

```
                      _ := (!chain-> [true ==> (dom rest - key Set.subset dom rest)   [dom-lemma-3]])
                     }
                 (!cases C
                   assume case1 := (k @ v = key @ val)
                     let {L := (!chain-> [(val =/= default rest)
                                        ==> (key in dom m)         [dom-lemma-1]])}
                      (!chain-> [case1
                             ==> (k = key & v = val)         [pair-axioms]
                             ==> (k = key)                   [left-and]
                             ==> (k in dom m)                [L]
                             ==> (k in dom m & ~ k in dom m) [augment]
                             ==> false                       [prop-taut]])
                   assume case2 := (k @ v in dmap->set rest - key)
                      (!chain-> [case2
                             ==> (k in dom rest - key)       [ms-lemma-1]
                             ==> (k in dom rest)             [Set.SC]
                             ==> (k in key Set.++ dom rest) [Set.in-lemma-3]
                             ==> (k in dom m)                [dom-def]
                             ==> (k in dom m & ~ k in dom m) [augment]
                             ==> false                       [prop-taut]]))))
}

conclude ms-lemma-1b' := (forall m k . ~ k in dom m ==> ~ exists v . k @ v in dmap->set m)
pick-any m:(DMap 'K 'V) k:'K
  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.lst->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 sm1 := [0 [['x --> 1] ['y --> 2] ['z --> 3]]]
define sm2 := [0 [['y --> 2] ['z --> 3] ['x --> 1]]]

(eval sm1 |^ ['z 'y])

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]])}
              (!chain-> [(k @ val in dmap->set m)
                     ==> (k @ v in dmap->set m)     [v=val]
                     ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]])
        assume case2 := (~ k in dom m)
          (!chain-> [case2
                 ==> (m at k = default m)                              [ms-theorem-2]
                 ==> (v = default m)                                   [hyp]
                 ==> (~ k in dom m & v = default m)                    [augment]
                 ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]]))

conclude at-characterization-2 :=
(forall m k v . k @ v in dmap->set m | ~ k in dom m & v = default m ==> m at k = v)
  pick-any m:(DMap 'K 'V) k:'K v:'V
    assume hyp := (k @ v in dmap->set m | ~ k in dom m & v = default m)
      (!cases hyp
        assume case1 := (k @ v in dmap->set m)
          (!chain-> [case1 ==> (m at k = v)   [ms-theorem-1]])
        assume case2 := (~ k in dom m & v = default m)
          (!chain-> [(~ k in dom m)
                 ==> (m at k = default m)   [ms-theorem-2]
                 ==> (m at k = v)           [(v = default m)]]))

conclude at-characterization :=
(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
    (!equiv
      (!chain [(m at k = v) ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [at-characterization-1]])
      (!chain [(k @ v in dmap->set m | ~ k in dom m & v = default m) ==> (m at k = v) [at-characterization-2]]))

define at-characterization-lemma :=
  (forall m k v . m at k = v & k in dom m ==> k @ v in dmap->set m)

define at-characterization-lemma-2 :=
  (forall m k v . m at k = v & v =/= default m ==> k @ v in dmap->set m)

(!force at-characterization-lemma)
(!force at-characterization-lemma-2)

}
```

```
EOF
(load "lib/basic/dmaps")


(load "c:\\np\\book\\fsetText")


open Pair

module DMap {

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

overload ++ update

```

```
set-precedence ++ 210

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]
define [h t] := [FSet.h FSet.t]

define [m m' m1 m2 rest] := [?m:(DMap 'S1 'S2) ?m':(DMap 'S3 'S4) ?m1:(DMap 'S5 'S6) ?m2:(DMap 'S7 'S8) ?rest:(DMap 'S

assert* apply-def :=
  [(empty-map d at _ = d)
   (k @ v ++ rest at x = v <== k = x)
   (k @ v ++ rest at x =  rest at x <== k =/= x)]

## Some testing:

define make-map :=
  lambda (L)
    match L {
       [] => (empty-map 0)
     | (list-of [x n] rest) => (update (x @ n) (make-map rest))
    }

define update* :=
  lambda (fm pairs)
    letrec {loop := lambda (pairs res)
                      match pairs {
                        [] => res
                      | (list-of [key val] more) => (loop more (update res key val))}}
       (loop pairs fm)


define f := lambda (i) [(string->id ("s" joined-with (val->string i))) i]

define L := (from-to 1 5)

define sample-map := (make-map (map f L))

# So sample-map maps 's1 to 1, ..., 's5 to 5.

define applied-to := apply

(eval sample-map at 's1)
(eval sample-map at 's2)
(eval sample-map at 's3)
(eval sample-map at 's4)
(eval sample-map at 's5)

# And this should give the default value 0:

(eval sample-map at 's99)

declare default: (K, V) [(DMap K V)] -> V [200 [alist->dmap]]

assert* default-def :=
  [(default empty-map d = d)
   (default _ ++ rest = default rest)]

(eval default sample-map)

declare remove: (S, T) [(DMap S T) S] -> (DMap S T) [- 120 [alist->dmap id]]

left-assoc -

assert* remove-def :=
  [(empty-map d - _ = empty-map d)
   ([key _] ++ rest - key = rest - key)
   (key =/= x ==> [key val] ++ rest - x = [key val] ++ (rest - x))]

declare dom: (S, T) [(DMap S T)] -> (FSet.Set S)   [[alist->dmap]]

assert* dom-def :=
```

```
  [(dom empty-map _ = null)
   (dom [k v] ++ rest = dom rest - k <== v = default rest)
   (dom [k v] ++ rest = k ++ dom rest <== v =/= default rest)]

declare size: (S, T) [(DMap S T)] -> N [[alist->dmap]]
assert* size-axioms := [(size m = card dom m)]

define rc1 := (forall m x . (m - x) at x = default m)

by-induction rc1 {
  (m as (empty-map d)) =>
     pick-any x
      (!chain [(m - x at x)
             = (m at x)      [remove-def]
             = d             [apply-def]
             = (default m)   [default-def]])
| (m as (update (pair k:'S v) rest)) =>
    let {IH := (forall x . rest - x at x = default rest)}
      pick-any x:'S
        (!two-cases
           assume (k = x)
            (!chain [(m - x at x)
                   = (m - k at k)   [(k = x)]
                   = (rest - k at k)   [remove-def]
                   = (default rest)    [IH]
                   = (default m)       [default-def]
                    ])
           assume (k =/= x)
             (!chain [(m - x at x)
                    = ((k @ v) ++ (rest - x) at x)  [remove-def]
                    = (rest - x at x)               [apply-def]
                    = (default rest)                [IH]
                    = (default m)                   [default-def]]))
}

define rc2 := (forall m k x . k =/= x ==> m - k at x = m at x)

by-induction rc2 {
 (m as (empty-map d:'V)) =>
   pick-any k:'K x:'K
     assume (k =/= x)
       let {L := (m - k at x);
            R := (m at x)}
         (!chain [L
                = (m at x)  [remove-def]])
| (m as (update (pair key:'K val:'V) rest:(DMap 'K 'V))) =>
   pick-any k:'K x:'K
     assume (k =/= x)
       let {IH := (forall k x . k =/= x ==> (rest - k) at x = rest at x)}
         (!two-cases
           assume (key = k)
            let {_ := (!by-contradiction (key =/= x)
                        (!chain [(key = x)
                             ==> (k = x)            [(key = k)]
                             ==> (k = x & k =/= x)  [augment]
                             ==> false              [prop-taut]]))}
            (!chain [(m - k at x)
                   = (((k @ val) ++ rest) - k at x)   [(key = k)]
                   = (rest - k at x)  [remove-def]
                   = (rest at x)      [IH]
                   = (m at x)         [apply-def]])
           assume (key =/= k)
             (!two-cases
                assume (x = key)
                  (!chain [(m - k at x)
                        = (([key val] ++ (rest - k)) at x)   [remove-def]
                        = (([x val] ++ (rest - k)) at x)   [(x = key)]
                        = val                              [apply-def]
                        = (([x val] ++ rest) at x)         [apply-def]
                        = (m at x)                          [(x = key)]])
```

```
                  assume (x =/= key)
                    (!chain [(m - k at x)
                         = (([key val] ++ (rest - k)) at x)   [remove-def]
                         = (rest - k at x)                     [apply-def]
                         = (rest at x)                         [IH]
                         = (m at x)                            [apply-def]])))
}

define rc3 := (forall m k . default m = default m - k)
by-induction rc3 {
  (m as (empty-map d:'V)) =>
    pick-any k
      (!chain [(default m)
            = (default m - k) [remove-def]])
 | (m as (update (pair key:'K val:'V) rest)) =>
    let {IH := (forall k . default rest = default rest - k)}
    pick-any k:'K
      (!two-cases
         assume (key = k)
          (!combine-equations
            (!chain [(default m)
                  = (default rest)      [default-def]
                  = (default rest - k)  [IH]])
            (!chain [(default m - k)
                  = (default rest - k)  [remove-def]]))
         assume (key =/= k)
           (!chain-> [(default m - k)
                   = (default key @ val ++ rest - k) [remove-def]
                   = (default rest - k)              [default-def]
                   = (default rest)                  [IH]
                   = (default m)                     [default-def]
                 ==> (default m - k = default m)
                 ==> (default m = default m - k)     [sym]]))
}

conclude dom-lemma-1 :=
  (forall k v rest . v =/= default rest ==> k in dom [k v] ++ rest)
pick-any k v rest
  assume hyp := (v =/= default rest)
  (!chain-> [true ==> (k in k ++ dom rest)     [FSet.in-lemma-1]
                ==> (k in dom [k v] ++ rest) [dom-def]])

conclude dom-lemma-2 :=
  (forall m k v . v =/= default m ==> dom m subset dom [k v] ++ m)
pick-any m k v
  assume hyp := (v =/= default m)
  (!FSet.subset-intro
     pick-any x
       (!chain [(x in dom m)
            ==> (x in k ++ dom m)      [FSet.in-lemma-3]
            ==> (x in dom [k v] ++ m)  [dom-def]]))

conclude dom-lemma-2b :=
  (forall m x k v . v =/= default m & x in dom m ==> x in dom [k v] ++ m)
pick-any m x k v
  assume (v =/= default m & x in dom m)
  let {_ := (!chain-> [(v =/= default m) ==> (dom m subset dom [k v] ++ m) [dom-lemma-2]])}
    (!chain-> [(x in dom m) ==> (x in dom [k v] ++ m) [FSet.SC]])

# conclude dom-lemma-2c :=
#   (forall m x k v . x in dom [k v] ++ m ==> x = k | x in dom m - k)
# pick-any m:(DMap 'K 'V) x:'K k:'K v:'V
#   assume hyp := (x in dom [k v] ++ m)
#     (!two-cases
#       assume (v = default m)
#         (!chain-> [hyp
#                ==> (x in dom m - k)          [dom-def]
#                ==> (x = k | x in dom m - k) [prop-taut]])
#       assume (v =/= default m)
#         (!chain-> [hyp
```

```
#                 ==> (x in k ++ dom m)          [dom-def]
#                 ==> (x = k | x in dom m - k)       [FSet.in-def]]))

define [< <=] := [N.< N.<=]
declare len: (S, T) [(DMap S T)] -> N [[alist->dmap]]

assert* len-def :=
  [(len empty-map _ = zero)
   (len _ @ _ ++ rest = S len rest)]

define len-lemma-1 :=
  (forall m k v . len m < len (k @ v) ++ m)

by-induction len-lemma-1 {
  (m as (empty-map d:'V)) =>
    pick-any k v
      let {len-left :=  (!chain [(len m) = zero                [len-def]]);
           len-right := (!chain [(len k @ v ++ m) = (S len m) [len-def]])}
       (!chain-> [true
              ==> (zero < S len m)          [N.Less.<-def]
              ==> (len m < len k @ v ++ m) [len-left len-right]])
 | (m as (update (pair key:'K val:'V) rest)) =>
     let {IH := (forall k v . len rest < len k @ v ++ rest)}
       pick-any k:'K v:'V
         let {len-left := (!chain [(len m)
                                  = (S len rest)  [len-def]]);
              len-right := (!chain [(len k @ v ++ m)
                                   = (S len m)      [len-def]
                                   = (S S len rest) [len-left]])}
          (!chain-> [true
                 ==> (S len rest < S S len rest)   [N.Less.<S]
                 ==> (len m < len k @ v ++ m)      [len-left len-right]])
}

conclude len-lemma-2 := (forall m k . len m - k <= len m)
by-induction len-lemma-2 {
  (m as (empty-map d:'V)) =>
    pick-any k
     (!chain-> [(len m - k)
              = (len m)               [remove-def]
          ==>  (len m - k <= len m) [N.Less=.<=-def]])
 | (m as (update (pair key:'K val:'V) rest)) =>
     pick-any k:'K
       let {IH := (forall k . len rest - k <= len rest);
            L2 := (!chain-> [true ==> (len rest - k <= len rest) [IH]]);
            L3 := (!chain-> [true ==> (len rest < len m)          [len-lemma-1]]);
            L4 := (!chain-> [L2 ==> (L2 & L3)                     [augment]
                                ==> (len rest - k < len m)        [N.Less=.transitive2]])}
        (!two-cases
          assume (key = k)
            (!chain-> [(len m - k)
                    = (len rest - k)                   [remove-def]
                  ==> (len m - k <= len rest - k)       [N.Less=.<=-def]
                  ==> (len m - k <= len rest - k & L2) [augment]
                  ==> (len m - k <= len rest)           [N.Less=.transitive]
                  ==> (len m - k <= len rest & L3)      [augment]
                  ==> (len m - k < len m)               [N.Less=.transitive2]
                  ==> (len m - k <= len m)              [N.Less=.<=-def]])
          assume (key =/= k)
            let {L5 := (!chain-> [(len m - k)
                                = (len [key val] ++ (rest - k)) [remove-def]
                                = (S len rest - k)              [len-def]])}

               (!chain-> [L4
                      ==> (S len rest - k <= len m)  [N.Less=.discrete]
                      ==> (len m - k <= len m)       [L5]]))
}

define len-lemma-3 :=
   (forall key val k rest . len rest - k < len key @ val ++ rest)
```

```
conclude len-lemma-3
  pick-any key:'K val:'V k:'K rest:(DMap 'K 'V)
    let {m := (key @ val ++ rest);
         L := (!chain-> [true
                     ==> (len rest - k <= len rest) [len-lemma-2]])}
      (!chain-> [true
             ==> (len rest < len m)          [len-lemma-1]
             ==> (L & len rest < len m)      [augment]
             ==> (len rest - k < len m)      [N.Less=.transitive2]])

transform-output eval [nat->int]

define (lemma-D-property m) :=
  (forall k .  k in dom m <==> m at k =/= default m)

define lemma-D := (forall m k . k in dom m <==> m at k =/= default m)

define lemma-D :=
  (forall m . lemma-D-property m)

(!strong-induction.measure-induction lemma-D len
pick-any m:(DMap 'K 'V)
  assume IH := (forall m' . len m' < len m ==> lemma-D-property m')
    conclude (lemma-D-property m)
      datatype-cases (lemma-D-property m) on m  {
        (em as (empty-map d:'V)) =>
         pick-any k
           (!equiv
             (!chain [(k in dom em)
                  ==> (k in null)    [dom-def]
                  ==> false          [FSet.NC]
                  ==> (em at k =/= default em) [prop-taut]])
             assume h := (em at k =/= default em)
               (!by-contradiction (k in dom em)
                 assume (~ k in dom em)
                   (!absurd (!reflex (default em))
                            (!chain-> [h ==> (d =/= default em)          [apply-def]
                                         ==> (default em =/= default em) [default-def]]))))
      | (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)]])}
            (!equiv
              assume hyp := (k in dom map)
                (!two-cases
                  assume (val = default rest)
                    let {L1 := (!by-contradiction (k =/= key)
                                 assume (k = key)
                                   (!absurd
                                    (!chain [(rest - key at key)
                                           = (default rest)   [rc1]
                                           = (default rest - key) [rc3]])
                                    (!chain-> [(k in dom map)
                                           ==> (key in dom map)         [(k = key)]
                                           ==> (key in dom rest - key) [dom-def]
                                           ==> (rest - key at key =/= default rest - key) [IH]])));
                         _ := (!ineq-sym L1)}
                      (!chain-> [(k in dom map)
                             ==> (k in dom rest - key)   [dom-def]
                             ==> (rest - key at k =/= default rest - key)   [IH]
                             ==> (rest - key at k =/= default rest)         [rc3]
                             ==> (rest - key at k =/= default map)          [default-def]
                             ==> (rest at k =/= default map)                [rc2]
                             ==> (map at k =/= default map)                [apply-def]])
                  assume case2 := (val =/= default rest)
                   let {M := method ()
                              (!chain-> [(map at k) = (map at key)      [(k = key)]
```

```
                                                   = val              [apply-def]
                                    ==> (map at k =/= default rest)   [case2]
                                    ==> (map at k =/= default map)    [default-def]])}
                   (!cases (!chain-> [hyp
                                   ==> (k in key ++ dom rest)    [dom-def]
                                   ==> (k = key | k in dom rest) [FSet.in-def]])
                      assume (k = key)
                        (!M)
                      assume (k in dom rest)
                        (!two-cases
                          assume (k = key)
                            (!M)
                          assume (k =/= key)
                            (!chain-> [(k in dom rest)
                                   ==> (rest at k =/= default rest) [IH]
                                   ==> (map at k  =/= default rest) [apply-def]
                                   ==> (map at k  =/= default map)  [default-def]]))))
              assume hyp := (map at k =/= default map)
                (!two-cases
                   assume case1 := (val = default rest)
                    let {k=/=key := (!by-contradiction (k =/= key)
                                       assume (k = key)
                                        let {p := (!chain [(map at k)
                                                         = (map at key)   [(k = key)]
                                                         = val            [apply-def]
                                                         = (default rest) [case1]
                                                         = (default map)  [default-def]])}
                                        (!absurd p hyp))}
                      (!chain-> [hyp
                             ==> (rest at k =/= default map) [apply-def]
                             ==> ((rest - key) at k =/= default map) [rc2]
                             ==> ((rest - key) at k =/= default rest) [default-def]
                             ==> ((rest - key) at k =/= default rest - key) [rc3]
                             ==> (k in dom rest - key)                      [IH]
                             ==> (k in dom map)                             [dom-def]])
                   assume case2 := (val =/= default rest)
                       (!two-cases
                         assume (k = key)
                           (!chain<- [(k in dom map)
                                  <== (key in dom map) [(k = key)]
                                  <== (key in key ++ dom rest) [dom-def]
                                  <== true                     [FSet.in-lemma-1]])
                         assume (k =/= key)
                           (!chain-> [hyp
                                  ==> (rest at k =/= default map)  [apply-def]
                                  ==> (rest at k =/= default rest) [default-def]
                                  ==> (k in dom rest)              [IH]
                                  ==> (k = key | k in dom rest)    [prop-taut]
                                  ==> (k in key ++ dom rest)       [FSet.in-def]
                                  ==> (k in dom map)               [dom-def]])))
       )
     })

conclude rc0 := (forall m x . ~ x in dom m - x)
  pick-any m:(DMap 'K 'V)  x:'K
    (!by-contradiction (~ x in dom m - x)
      assume hyp := (x in dom m - x)
       (!absurd (!chain-> [true ==> (m - x at x = default m) [rc1]])
                (!chain-> [hyp
                       ==> (m - x at x =/= default m - x)   [lemma-D]
                       ==> (m - x at x =/= default m)       [rc3]])))

conclude dom-lemma-3 := (forall m k . dom (m - k) subset dom m)
pick-any m:(DMap 'K 'V) k:'K
(!FSet.subset-intro
  pick-any x:'K
    assume hyp := (x in dom m - k)
      (!two-cases
         assume (x = k)
          let {L := (!chain-> [true ==> (m - k at k = default m) [rc1]])}
```

```
              (!chain-> [hyp
                     ==> (k in dom m - k)                   [(x = k)]
                     ==> (m - k at k =/= default m - k)  [lemma-D]
                     ==> (m - k at k =/= default m)      [rc3]
                     ==> (L & m - k at k =/= default m)  [augment]
                     ==> false                           [prop-taut]
                     ==> (x in dom m)                    [prop-taut]])
          assume (x =/= k)
            (!chain-> [hyp
                   ==> (m - k at x =/= default m - k)   [lemma-D]
                   ==> (m at x =/= default m - k)       [rc2]
                   ==> (m at x =/= default m)           [rc3]
                   ==> (x in dom m)                     [lemma-D]])))

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)              [FSet.SC]
              ==> (k = key | k in dom rest)    [prop-taut]
              ==> (k in key ++ dom rest)       [FSet.in-def]
              ==> (k in dom [key val] ++ rest) [dom-def]]))

declare dmap->set: (K, V) [(DMap K V)] -> (FSet.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 FSet.null)     [dmap->set-def]
                        ==> false                    [FSet.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) [FSet.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)                           [FSet.SC]
                          ==> (k in key ++ dom rest)                              [FSet.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)              [FSet.SC]
#               ==> (k = key | k in dom rest)    [prop-taut]
#               ==> (k in key ++ dom rest)       [FSet.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) (FSet.Set S)] -> Boolean
                         [[alist->dmap alist->dmap FSet.lst->set]]


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

let {m1 := [77 [['x --> 1] ['y --> 2]]];
     m2 := [78 [['y --> 2] ['x --> 1]]]}
 (eval (agree-on m1 m2 ['x 'y]))

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 FSet.null:(FSet.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                 [FSet.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 (FSet.insert h:'K t:(FSet.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)  [FSet.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)            [FSet.in-lemma-1]
                                     ==> (m1 at h = m2 at h) [hyp]]);
                        L2 := pick-any k:'K
                                (!chain [(k in t)
                                     ==> (k in A)             [FSet.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)
}

define AGC := agreement-characterization

conclude downward-agreement-lemma :=
  (forall B A m1 m2 . (agree-on m1 m2 A) & B subset A ==> (agree-on m1 m2 B))
pick-any B:(FSet.Set 'K) A:(FSet.Set 'K) m1:(DMap 'K 'V) m2:(DMap 'K 'V)
  assume hyp := ((agree-on m1 m2 A) & B subset A)
    let {L := pick-any k:'K
                assume hyp := (k in B)
                  (!chain-> [hyp
                         ==> (k in A) [FSet.SC]
                         ==> (m1 at k = m2 at k)  [AGC]])}
      (!chain-> [L ==> (agree-on m1 m2 B) [AGC]])

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

by-induction ms-lemma-1b {
  (m as (empty-map d:'V)) =>
     pick-any k
      assume hyp := (~ k in dom m)
        pick-any v:'V
          (!by-contradiction (~ k @ v in dmap->set m)
             (!chain [(k @ v in dmap->set m)
                  ==> (k @ v in FSet.null)      [dmap->set-def]
                  ==> false                     [FSet.NC]]))
| (m as (update (pair key:'K val:'V) rest)) =>
    let {IH :=  (forall k . ~ k in dom rest ==> forall v . ~ k @ v in dmap->set rest)}
      pick-any k
        assume hyp := (~ k in dom m)
          pick-any v:'V
            (!by-contradiction (~ k @ v in dmap->set m)
               assume sup := (k @ v in dmap->set m)
               (!two-cases
                assume (val = default rest)
                  (!chain-> [sup
                         ==> (k @ v in dmap->set rest - key)  [dmap->set-def]
                         ==> (k in dom rest - key)            [ms-lemma-1]
                         ==> (k in dom m)                     [dom-corrolary-1]
                         ==> (k in dom m & hyp)               [augment]
                         ==> false                            [prop-taut]])
                assume (val =/= default rest)
                let {C :=
                      (!chain-> [sup
                             ==> (k @ v in key @ val FSet.++ dmap->set rest - key)    [dmap->set-def]
                             ==> (k @ v = key @ val | k @ v in dmap->set rest - key)  [FSet.in-def]]);
                     _ := (!chain-> [true ==> (dom rest - key FSet.subset dom rest)   [dom-lemma-3]])
```

```
                  }
                (!cases C
                  assume case1 := (k @ v = key @ val)
                    let {L := (!chain-> [(val =/= default rest)
                                   ==> (key in dom m)          [dom-lemma-1]])}
                     (!chain-> [case1
                            ==> (k = key & v = val)          [pair-axioms]
                            ==> (k = key)                    [left-and]
                            ==> (k in dom m)                 [L]
                            ==> (k in dom m & ~ k in dom m)  [augment]
                            ==> false                        [prop-taut]])
                  assume case2 := (k @ v in dmap->set rest - key)
                     (!chain-> [case2
                            ==> (k in dom rest - key)        [ms-lemma-1]
                            ==> (k in dom rest)              [FSet.SC]
                            ==> (k in key FSet.++ dom rest)  [FSet.in-lemma-3]
                            ==> (k in dom m)                 [dom-def]
                            ==> (k in dom m & ~ k in dom m)  [augment]
                            ==> false                        [prop-taut]]))))
}

conclude ms-lemma-1b' := (forall m k . ~ k in dom m ==> ~ exists v . k @ v in dmap->set m)
pick-any m:(DMap 'K 'V) k:'K
  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) (FSet.Set S)] -> (DMap S T) [150 |^ [alist->dmap FSet.lst->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 sm1 := [0 [['x --> 1] ['y --> 2] ['z --> 3]]]
define sm2 := [0 [['y --> 2] ['z --> 3] ['x --> 1]]]

(eval sm1 |^ ['z 'y])

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 FSet.null)     [dmap->set-def]
                      ==> false                    [FSet.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)) [FSet.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 FSet.null)   [dom-def]
               ==> false              [FSet.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)          [FSet.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')           [FSet.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) [FSet.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') [FSet.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')           [FSet.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 := (!FSet.subset-intro
                  pick-any k:'K
                    assume (k in R)
                      (!cases (!chain-> [(k in R)
                                     ==> (k = key | k in dom m - key)  [FSet.in-def]])
                        assume (k = key)
                          (!chain-> [true
                                 ==> (key in key ++ dom m)        [FSet.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)         [FSet.SC]
                                 ==> (k in key ++ dom m) [FSet.in-lemma-3]
                                 ==> (k in L)               [dom-def]])));
       L->R := (!FSet.subset-intro
                  pick-any k:'K
                    assume (k in L)
                       let {M := method ()
                                   (!chain-> [true
                                          ==> (key in key ++ dom m - key)  [FSet.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) [FSet.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)                      [FSet.in-def]]))))}
    (!FSet.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 FSet.null)  [dom-def]
                       ==> false             [FSet.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) [FSet.in-def]])
                    assume case1 := (k = key)
                      (!chain-> [true
                             ==> (key @ val in key @ val ++ dmap->set rest - key)  [FSet.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) [FSet.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]])}
              (!chain-> [(k @ val in dmap->set m)
                     ==> (k @ v in dmap->set m)    [v=val]
                     ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]])
        assume case2 := (~ k in dom m)
          (!chain-> [case2
                 ==> (m at k = default m)                              [ms-theorem-2]
                 ==> (v = default m)                                   [hyp]
                 ==> (~ k in dom m & v = default m)                    [augment]
                 ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [prop-taut]]))

conclude at-characterization-2 :=
(forall m k v . k @ v in dmap->set m | ~ k in dom m & v = default m ==> m at k = v)
  pick-any m:(DMap 'K 'V) k:'K v:'V
    assume hyp := (k @ v in dmap->set m | ~ k in dom m & v = default m)
      (!cases hyp
        assume case1 := (k @ v in dmap->set m)
          (!chain-> [case1 ==> (m at k = v)   [ms-theorem-1]])
        assume case2 := (~ k in dom m & v = default m)
          (!chain-> [(~ k in dom m)
                 ==> (m at k = default m)   [ms-theorem-2]
                 ==> (m at k = v)           [(v = default m)]]))

conclude at-characterization :=
(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
    (!equiv
      (!chain [(m at k = v) ==> (k @ v in dmap->set m | ~ k in dom m & v = default m) [at-characterization-1]])
      (!chain [(k @ v in dmap->set m | ~ k in dom m & v = default m) ==> (m at k = v) [at-characterization-2]]))

define at-characterization-lemma :=
  (forall m k v . m at k = v & k in dom m ==> k @ v in dmap->set m)

define at-characterization-lemma-2 :=
  (forall m k v . m at k = v & v =/= default m ==> k @ v in dmap->set m)

(!force at-characterization-lemma)
(!force at-characterization-lemma-2)

} # module DMap

EOF
```

```
(load "c:\\np\\book\\dmapText2")

#START_LOAD

define map-identity-characterization-1 :=
  (forall m1 m2 . (forall k . m1 at k = m2 at k) ==> m1 = m2)

conclude map-identity-characterization-1
  pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
    assume hyp := (forall k . m1 at k = m2 at k)
      let {L1 := conclude (dmap->set m1 = dmap->set m2)
                  (!FSet.set-identity-intro-direct
                     (!pair-converter
                        pick-any k:'K v:'V
                          (!equiv
                              assume hyp' := (k @ v in dmap->set m1)
                              let {L := (!chain-> [hyp ==> (k in dom m1) [ms-lemma-1]])}
                              (!chain-> [(k @ v in dmap->set m1)
                                     ==> (m1 at k = v)              [at-characterization]
                                     ==> (m1 at k = v & L)          [augment]
                                     ==> (m2 at k = v & L)          [hyp]
                                     ==> (m2 at k = v)              [hyp]
                                     ==> (k @ v in dmap->set m2)    [at-characterization]])
                              (!chain [(k @ v in dmap->set m2)
                                   ==> (m2 at k = v)              [at-characterization]
                                   ==> (m1 at k = v)              [hyp]
                                   ==> (k @ v in dmap->set m1)    [at-characterization]]))))

           ;
           L2 := conclude (default m1 = default m2)
                   (!force (default m1 = default m2))
           ; _ := (halt)
           }
       (!chain-> [L2
              ==> (L2 & L1)   [augment]
              ==> (m1 = m2)   [dmap-identity]])
#END_LOAD
(load "c:\\np\\book\\dmapText2")

define identity-result-1 :=
  (forall m1 m2 . (forall k . m1 at k = m2 at k) ==>
                  dmap->set m1 = dmap->set m2 & default m1 = default m2)


#START_LOAD

conclude identity-result-1
  pick-any m1:(DMap 'K 'V) m2:(DMap 'K 'V)
    assume hyp := (forall k . m1 at k = m2 at k)
      let {[s1 s2] := [(dmap->set m1) (dmap->set m2)];
           L1 :=
            (!by-contradiction (s1 = s2)
              assume hyp' := (s1 =/= s2)
                (!cases (!chain-> [hyp' ==> ((exists p . p in s1 & ~ p in s2) |
                                             (exists p . p in s2 & ~ p in s1))
                                             [FSet.neg-set-identity-characterization-2]])
                  assume case1 := (exists p . p in s1 & ~ p in s2)
                    let {case1 := conclude (exists k v . k @ v in s1 & ~ k @ v in s2)
                                    (!pair-converter-2 case1)}
                    pick-witnesses k v for case1

                  assume case2 := (exists p . p in s2 & ~ p in s1)
                      (!dhalt)))}
      (!dhalt)
#END_LOAD
# (load "c:\\np\\book\\dmapText")
```