Formalization of Weighted Sets

Mathias Schack Rabing Dmitriy Traytel

June 25, 2026

Abstract

We define *weighted sets* as a generalization of finite multisets where multiplicities are replaced by weights from a *refinable* [1] *abelian semigroup*. Weighted sets are equivalently represented as functions from elements to optional weights returning `None` for all but finitely many elements, or as quotients of element-weight lists modulo permutation and regrouping. We register weighted sets as a bounded natural functor (BNF), enabling nested (co)recursion through them in (co)datatypes.

Contents

theory *Weighted-Set*
imports
HOL−Library.Multiset
begin

1 Algebraic Preliminaries

instantiation *option* :: (*ab-semigroup-add*) *comm-monoid-add* **begin**
definition *zero-option* **where** *zero-option* = *None*
definition *plus-option* **where** *plus-option a b* = (*case* (*a*, *b*) *of* (*Some x*, *Some y*) ⇒ *Some* (*x* + *y*) | (*Some x*, *None*) ⇒ *Some x* | (*None*, *Some x*) ⇒ *Some x* | - ⇒ *None*)
instance
by *standard* (*auto simp*: *zero-option-def plus-option-def ac-simps split*: *option.splits*)
end

The notion of refinability is due to Gumm and Schröder [1], who introduced it for monoids. We generalize it to semigroups.

class *ref-ab-semigroup-add* = *ab-semigroup-add* +
assumes *refinable*: (*a* :: 'a :: *ab-semigroup-add*) + *b* = *c* + *d* ⟹
(∃ (*e11* :: ('a :: *ab-semigroup-add*) *option*) *e12 e21 e22*. *Some a* = *e11* + *e12* ∧
Some b = *e21* + *e22* ∧ *Some c* = *e11* + *e21* ∧ *Some d* = *e12* + *e22*)

lemma *plus-option-simps* [*simp*]: *a* + *None* = *a None* + *a* = *a Some c* + *Some d* = *Some* (*c* + *d*)
unfolding *add.right-neutral add.left-neutral zero-option-def*[*symmetric*] *atomize-conj*
unfolding *plus-option-def*
by *simp*

lemma *plus-option-case*: *Some e* + *f* = *Some* (*case f of Some f'* ⇒ *e* + *f'* | *None* ⇒ *e*) *f* + *Some e* = *Some* (*case f of Some f'* ⇒ *f'* + *e* | *None* ⇒ *e*)
unfolding *add.right-neutral add.left-neutral zero-option-def*[*symmetric*] *atomize-conj*
unfolding *plus-option-def*
by(*cases f*; *simp*)

instantiation *option* :: (*ref-ab-semigroup-add*) *ref-ab-semigroup-add* **begin**
instance
proof *intro-classes*
fix *a b c d* :: 'a *option*
assume *A1*: *a* + *b* = *c* + *d*
show ∃ *e11 e12 e21 e22*. *Some a* = *e11* + *e12* ∧ *Some b* = *e21* + *e22* ∧ *Some c* = *e11* + *e21* ∧ *Some d* = *e12* + *e22*
proof −
consider (*Some*) *a' b' c' d'* **where** *a* = *Some a' b* = *Some b' c* = *Some c' d* = *Some d'*
| (*a*) *a* = *None* | (*b*) *b* = *None* | (*c*) *c* = *None* | (*d*) *d* = *None*
by *fastforce*
then show *?thesis*
proof (*cases*)

```
    case Some
    then have a' + b' = c' + d'
      using A1 unfolding plus-option-def by auto
    from refinable[OF this] show ?thesis
      unfolding Some by (metis plus-option-simps(3))
  next
    case a
    with A1 show ?thesis by (metis plus-option-simps(1,2,3))
  next
    case b
    with A1 show ?thesis by (metis plus-option-simps(1,3))
  next
    case c
    with A1 show ?thesis by (metis plus-option-simps(1,2,3))
  next
    case d
    with A1 show ?thesis by (metis plus-option-simps(1,3))
  qed
 qed
qed
end
```

2 The Positive Representation

```
abbreviation sum-key where
  sum-key kxs e ≡ fold (λ(-,w) y. Some w + y) (filter (λ(e',-). e = e') kxs) None

definition eq-wset where
   eq-wset (kxs :: ('a × ('w :: ref-ab-semigroup-add)) list) (kys :: ('a × ('w ::
ref-ab-semigroup-add)) list) =
  (∀ e. sum-key kxs e = sum-key kys e)

declare [[typedef-overloaded]]

quotient-type ('a, 'w) wset = ('a × 'w :: ref-ab-semigroup-add) list / eq-wset
  by(auto intro!: equivpI reflpI sympI transpI simp: eq-wset-def)

lemma get-abs-wset: ∃ l. M = abs-wset l
  by (metis Quotient3-abs-rep Quotient3-wset)

lemma fold-Some: fold (λ(a, w). (+) (Some w)) xs (Some w) ≠ None None ≠
fold (λ(a, w). (+) (Some w)) xs (Some w)
proof−
  have H: fold (λ(a, w). (+) (Some w)) xs (Some w) ≠ None
  proof(induction xs arbitrary: w)
    case Nil
    then show ?case
      by simp
  next
```

```
    case (Cons x xs)
    then show ?case
      by(cases x; simp)
  qed
  show fold (λ(a, w). (+) (Some w)) xs (Some w) ≠ None
    by(rule H)
  show None ≠ fold (λ(a, w). (+) (Some w)) xs (Some w)
    using H
    by force
qed

lemma fold-Some': ∃ w'. fold (λ(a, w). (+) (Some w)) xs (Some w) = Some w'
  using fold-Some
  by auto


lemma fold-Some-out: fold (λ(a, w). (+) (Some w)) xs (Some w) = (Some w) +
fold (λ(a, w). (+) (Some w)) xs None
proof(induction xs arbitrary: w)
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
  then show ?case
    by(cases x; simp add: add.assoc[symmetric] add.commute)
qed


lemma eq-wset-fst:
  assumes H: eq-wset xs xs'
  shows set (map fst xs') = set (map fst xs)
proof −
  have fold-eq: ∀ a. (sum-key xs a = None) = (sum-key xs' a = None)
    using H
    unfolding eq-wset-def
    by presburger
  have fold-True: fold (λ- -. True) L True for L :: 'c list
    by(induction L; simp)
  have fold-None-set: (sum-key xs a = None) = (a ∉ set (map fst xs)) for a :: 'a
and xs :: ('a × 'b) list
  proof (induction xs)
    case Nil
    show ?case by simp
  next
    case (Cons h t)
    obtain h1 h2 where h-def: h = (h1, h2) by (cases h)
    then show ?case using Cons.IH by (simp add: fold-Some fold-True)
  qed
```

```
  have ∀ a. (a ∉ set (map fst xs)) = (a ∉ set (map fst xs'))
    using fold-eq fold-None-set
    by metis
  then show ?thesis
    by blast
qed

lemma eq-wset-refl[simp]: eq-wset xs xs
  unfolding eq-wset-def
  by simp

lemma eq-wset-sym: eq-wset xs xs' ⟹ eq-wset xs' xs
  unfolding eq-wset-def
  by fastforce

lemma eq-wset-trans: eq-wset xs ys ⟹ eq-wset ys zs ⟹ eq-wset xs zs
  unfolding eq-wset-def
  by fastforce

lemma eq-wset-elem-switch: eq-wset (x # x' # xs) (x' # x # xs)
  unfolding eq-wset-def
  by(auto simp add: add.commute)

lemma eq-wset-elem-comb: eq-wset ((x,w) # (x,w') # xs) ((x,w + w') # xs)
  unfolding eq-wset-def
  by(auto simp add: add.commute)

lemma fold-elem-back-aux: fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a')
(xs @ e1 # e2 # x's)) w =
        fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a') (xs @ e2 # e1 #
x's)) w
  for a :: 'c and e1 e2 :: 'c × ('d :: ab-semigroup-add) and xs x's :: ('c × 'd) list
and w :: 'd option
 by(induction xs; (auto simp add: add.assoc[symmetric], simp only: add.commute))

lemma fold-elem-back: fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a') (xs
@ e # x's)) w =
       fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a') (xs @ x's @ [e])) w
proof (induction x's arbitrary: xs w)
  case Nil
  show ?case by simp
next
  case (Cons h t)
  from this[of xs @ [h] w] show ?case by (subst fold-elem-back-aux) auto
qed

lemma eq-wset-elem-back: eq-wset (xs @ e # x's) (xs @ x's @ [e])
  unfolding eq-wset-def
  using fold-elem-back
```

by *fast*

lemma *fold-elem-back'*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *a* = *a'*) (*e* # *x's*)) *w* =
fold (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *a* = *a'*) (*x's* @ [*e*])) *w*
by (*metis append-self-conv2 fold-elem-back*)

lemma *eq-wset-elem-back'*: *eq-wset* (*e* # *x's*) (*x's* @ [*e*])
by (*metis eq-wset-def fold-elem-back'*)

lemma *fold-Some-back*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *a* = *a'*) *xs*) (*Some* (*M* :: - :: *ref-ab-semigroup-add*)) = *sum-key* (*xs* @ [(*a*, *M*)]) *a*
proof −
have *fold-Some-front*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *a* = *a'*) *xs*) (*Some M*) = *sum-key* ((*a*, *M*) # *xs*) *a*
by *simp*
show *?thesis*
using *eq-wset-elem-back'*
unfolding *eq-wset-def fold-Some-front*
by *metis*
qed

lemma *fold-back'*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *a* = *a'*) *xs*) (*Some* (*M* :: - :: *ref-ab-semigroup-add*) + *w*) = *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *a* = *a'*) (*xs* @ [(*a*,*M*)])) *w*
proof −
have *fold-Some-front'*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *a* = *a'*) *xs*) (*Some M* + *w*) = *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *a* = *a'*) ((*a*,*M*) # *xs*)) *w*
by *simp*
show *?thesis*
using *fold-elem-back'*
unfolding *eq-wset-def fold-Some-front'*
by *metis*
qed

lemma *eq-wset-append*: *eq-wset xs xs'* ⟹ *eq-wset ys ys'* ⟹ *eq-wset* (*xs* @ *ys*) (*xs'* @ *ys'*)
proof −
assume *H1*: *eq-wset xs xs'* **and** *H2*: *eq-wset ys ys'*
have *xs-eq*: ∀ *a*. *sum-key xs a* = *sum-key xs' a*
using *H1* **unfolding** *eq-wset-def* **by** *simp*
have *ys-eq*: ∀ *a*. *sum-key ys a* = *sum-key ys' a*
using *H2* **unfolding** *eq-wset-def* **by** *simp*
show *eq-wset* (*xs* @ *ys*) (*xs'* @ *ys'*)
unfolding *eq-wset-def*
proof (*rule allI*)
fix *a*
have *ha-xs*: *sum-key xs a* = *sum-key xs' a* **using** *xs-eq* **by** *simp*

```
    have ha-ys: sum-key ys a = sum-key ys' a using ys-eq by simp
    show sum-key (xs @ ys) a = sum-key (xs' @ ys') a
    proof (cases sum-key xs' a)
      case None
      show ?thesis using None ha-xs ha-ys by simp
    next
      case (Some w)
      show ?thesis using Some ha-xs ha-ys by (simp add: fold-Some-back)
    qed
  qed
qed

lemma eq-wset-elem-remove: eq-wset xs xs' ⟹ eq-wset (e # xs) (e # xs')
  by (metis append-Cons append-Nil eq-wset-refl eq-wset-append)

lemma eq-wset-append-sym: eq-wset (xs @ ys) (ys @ xs)
proof (induction xs)
  show eq-wset ([] @ ys) (ys @ [])
    by simp
next
  fix a :: 'a × 'b
    and xs :: ('a × 'b) list
  assume ind: eq-wset (xs @ ys) (ys @ xs)
  have eq-wset (a # xs @ ys) (xs @ ys @ [a])
    by (metis append-assoc eq-wset-elem-back')
  also have eq-wset (ys @ a # xs) (ys @ xs @ [a])
    by (metis eq-wset-elem-back)
  ultimately have eq-wset-append-sym-aux: eq-wset (xs @ ys @ [a]) (ys @ xs @
[a]) ⟹ eq-wset ((a # xs) @ ys) (ys @ a # xs)
    by (metis append-Cons eq-wset-trans eq-wset-sym)
  show eq-wset ((a # xs) @ ys) (ys @ a # xs)
    using eq-wset-append[OF ind eq-wset-refl[of [a]]]
    by (intro eq-wset-append-sym-aux) simp
qed
```

3 Basic Operations

```
lift-definition wempty :: ‹('a, 'w :: ref-ab-semigroup-add) wset› is
  ‹[]› .

lift-definition weight :: ‹('a, 'w :: ref-ab-semigroup-add) wset ⇒ 'a ⇒ 'w option›
is
  ‹λkxs e. sum-key kxs e›
  unfolding eq-wset-def
  by simp

lemma weight-eq-iff: (∀ x. weight M x = weight N x) = (M = N)
  using get-abs-wset[of M] get-abs-wset[of N]
  by(auto simp add: weight.abs-eq wset.abs-eq-iff eq-wset-def)
```

lift-definition *wsingle* :: ‹*'a* ⇒ *'w* ⇒ (*'a*, *'w* :: *ref-ab-semigroup-add*) *wset*› **is**
‹λ*a w*. [(*a*,*w*)]› **.**

lift-definition *wset* :: ‹(*'a*, *'w* :: *ref-ab-semigroup-add*) *wset* ⇒ *'a set*› **is**
‹λ*M*. *set* (*map fst M*)› **by**(*drule eq-wset-fst*)(*auto simp add*: *image-def*)

lift-definition *wadd* :: ‹(*'a*, *'w* :: *ref-ab-semigroup-add*) *wset* ⇒ (*'a*, *'w*) *wset* ⇒ (*'a*, *'w*) *wset*› **is**
‹λ*M1 M2*. *M1* @ *M2*› **using** *eq-wset-append* **by** *metis*

lemma *sum-key-wupdate-same*:
sum-key (*case w of None* ⇒ *filter* (λ(*x'*, -). *x* ≠ *x'*) *l*
| *Some w'* ⇒ (*x*, *w'*) # *filter* (λ(*x'*, -). *x* ≠ *x'*) *l*) *x* = *w*
for *l* :: (*'a* × (*'w* :: *ref-ab-semigroup-add*)) *list*
by (*cases w*) (*simp-all add*: *case-prod-beta*)

lemma *sum-key-wupdate-diff*:
x ≠ *x'* ⟹
sum-key (*case w of None* ⇒ *filter* (λ(*x''*, -). *x* ≠ *x''*) *l*
| *Some w'* ⇒ (*x*, *w'*) # *filter* (λ(*x''*, -). *x* ≠ *x''*) *l*) *x'* = *sum-key l x'*
for *l* :: (*'a* × (*'w* :: *ref-ab-semigroup-add*)) *list*
proof (*cases w*)
case *None*
assume *neq*: *x* ≠ *x'*
have *filter-eq*: *filter* (λ*y*. *x* ≠ *fst y* ∧ *x'* = *fst y*) *l* = *filter* (λ(*e'*, -). *x'* = *e'*) *l* ∧
filter (λ*y*. *x'* = *fst y* ∧ *x* ≠ *fst y*) *l* = *filter* (λ(*e'*, -). *x'* = *e'*) *l*
using *neq* **by** (*auto intro*!: *filter-cong simp*: *case-prod-beta*)
then show *?thesis*
unfolding *None* **by** (*simp add*: *case-prod-beta filter-eq*)
next
case (*Some w'*)
assume *neq*: *x* ≠ *x'*
have *filter-eq*: *filter* (λ*y*. *x* ≠ *fst y* ∧ *x'* = *fst y*) *l* = *filter* (λ(*e'*, -). *x'* = *e'*) *l* ∧
filter (λ*y*. *x'* = *fst y* ∧ *x* ≠ *fst y*) *l* = *filter* (λ(*e'*, -). *x'* = *e'*) *l*
using *neq* **by** (*auto intro*!: *filter-cong simp*: *case-prod-beta*)
then show *?thesis*
unfolding *Some* **using** *neq*
by (*simp add*: *case-prod-beta filter-eq*)
qed

lift-definition *wupdate* :: ‹(*'a*, *'w* :: *ref-ab-semigroup-add*) *wset* ⇒ *'a* ⇒ *'w option* ⇒ (*'a*, *'w*) *wset*› **is**
‹λ*M x w*. *case w of Some w'* ⇒ (*x*,*w'*) # (*filter* (λ(*x'*,-). *x* ≠ *x'*) *M*) | *None* ⇒ *filter* (λ(*x'*,-). *x* ≠ *x'*) *M*›
proof −
fix *l1 l2* :: (*'a* × (*'w* :: *ref-ab-semigroup-add*)) *list* **and** *a* :: *'a* **and** *w* :: *'w option*
assume *hyp*: *eq-wset l1 l2*
show *eq-wset* (*case w of Some w'* ⇒ (*a*, *w'*) # *filter* (λ(*x'*, -). *a* ≠ *x'*) *l1*

```
                | None ⇒ filter (λ(x', -). a ≠ x') l1)
          (case w of Some w' ⇒ (a, w') # filter (λ(x', -). a ≠ x') l2
                | None ⇒ filter (λ(x', -). a ≠ x') l2)
    unfolding eq-wset-def
  proof
    fix a'
    show sum-key (case w of Some w' ⇒ (a, w') # filter (λ(x', -). a ≠ x') l1
                  | None ⇒ filter (λ(x', -). a ≠ x') l1) a' =
      sum-key (case w of Some w' ⇒ (a, w') # filter (λ(x', -). a ≠ x') l2
                  | None ⇒ filter (λ(x', -). a ≠ x') l2) a'
    proof (cases a = a')
      case True
      then show ?thesis by (simp only: True sum-key-wupdate-same)
    next
      case False
      with hyp show ?thesis
        unfolding eq-wset-def
        by (simp add: sum-key-wupdate-diff)
    qed
  qed
qed

instantiation wset :: (type, type) size
begin

definition size-wset where
  size-wset-overloaded-def: size-wset M = card (wset M)
instance ..

end

lemma weight-wsingle[simp] : weight (wsingle x w) x' = (if x = x' then Some w
else None)
  by transfer simp

lemma sum-key-append-aux:
  sum-key (l1 @ l2) x = sum-key l1 x + sum-key l2 x
  for l1 l2 :: ('a × ('w :: ref-ab-semigroup-add)) list
  by (induction l1) (auto simp add: fold-Some-back fold-back' add.assoc)

lemma weight-add[simp] : weight (wadd M1 M2) x = weight M1 x + weight M2 x
  by transfer (rule sum-key-append-aux)

lemma weight-wempty[simp] : weight wempty = (λ- . None)
  by transfer simp

lemma weight-wupdate[simp] : weight (wupdate M x w) = (weight M)(x := w)
proof −
  have H1: weight (wupdate M x w) x = w
```

by *transfer* (*rule sum-key-wupdate-same*)
have *H2*: $x \neq x' \Longrightarrow$ *weight* (*wupdate M x w*) $x' =$ *weight M x'* **for** x'
by *transfer* (*rule sum-key-wupdate-diff*)
show *?thesis*
proof (*rule ext*)
fix x'
show *weight* (*wupdate M x w*) $x' =$ ((*weight M*)($x :=$ *w*)) x'
proof (*cases* $x = x'$)
case *True*
then show *?thesis* **using** *H1* **by** *simp*
next
case *False*
then show *?thesis* **using** *H2*[*OF False*] **by** *simp*
qed
qed
qed

lemma *wupdate-wupdate*[*simp*]: *wupdate* (*wupdate M x w*) *x w'* = *wupdate M x w'*
by(*simp add*: *weight-eq-iff*[*symmetric*])

4 The Splitting Relation

abbreviation *fold'* **where**
fold' l ≡ *foldl* (+) (*hd l*) (*tl l*)

inductive *list-split* :: ('*a*×'*w* :: *ref-ab-semigroup-add*) *list* ⇒ ('*a*×'*w*) *list* ⇒ *bool* **where**
Base: *list-split* [] []
| *Split*: *list-split xs'' ys* ⟹ *xs* = *xs'* @ *xs''* ⟹ *w* = *fold'* (*map snd xs'*) ⟹ *xs'* ≠ [] ⟹ *list-all* (λ(*a*,*b*). *a* = *y*) *xs'* ⟹ *list-split xs* ((*y*,*w*) # *ys*)

inductive *list-split'* :: ((('*w* :: *ref-ab-semigroup-add*) *option*) *list*) *list* ⇒ ('*w option*) *list* ⇒ *bool* **where**
Base': *list-split'* [] []
| *Split'*: *list-split' xss ys* ⟹ *y* = *fold' xs* ⟹ *xs* ≠ [] ⟹ *list-split'* (*xs* # *xss*) (*y* # *ys*)

lemma *list-split-cons-eq*: *list-split xs1 xs2* ⟹ *list-split* (*x* # *xs1*) (*x* # *xs2*)
proof (*cases x*)
case (*Pair x1 w*)
assume *H*: *list-split xs1 xs2*
show *?thesis*
unfolding *Pair*
by (*rule Split*[**where** *xs''* = *xs1* **and** *ys* = *xs2* **and** *xs'* = [(*x1*, *w*)] **and** *y* = *x1* **and** *w* = *w*])
(*use H* **in** *simp-all*)
qed

lemma *list-split-refl*[*simp*]: *list-split xs xs*
proof (*induction xs*)
  case *Nil*
  show *?case* **by** (*subst list-split.simps*; *simp*)
next
  case (*Cons x xs'*)
  show *?case* **by** (*rule list-split-cons-eq*; *simp add*: *Cons.IH*)
qed

lemma *list-split-comb*: *list-split* ((*x*, *w*) # (*x*, *w'*) # *xs*) ((*x*, *w* + *w'*) # *xs*)
proof (*rule Split*[**where** *xs''* = *xs* **and** *ys* = *xs* **and** *xs'* = [(*x*, *w*), (*x*, *w'*)] **and** *y* = *x* **and** *w* = *w* + *w'*])
  show *list-split xs xs* **by** *simp*
  show (*x*, *w*) # (*x*, *w'*) # *xs* = [(*x*, *w*), (*x*, *w'*)] @ *xs* **by** *simp*
  show *w* + *w'* = *fold'* (*map snd* [(*x*, *w*), (*x*, *w'*)]) **by** *simp*
  show [(*x*, *w*), (*x*, *w'*)] ≠ [] **by** *simp*
  show *list-all* (λ(*a*, *b*). *a* = *x*) [(*x*, *w*), (*x*, *w'*)] **by** *simp*
qed

lemma *list-split-nil*[*simp*]: *list-split xs* [] = (*xs* = []) *list-split* [] *xs* = (*xs* = [])
  by(*subst list-split.simps*; *simp*)+

lemma *eq-wset-nil*[*simp*]: *eq-wset xs* [] = (*xs* = [])
proof (*cases xs*)
  case *Nil*
  then show *?thesis* **by** *simp*
next
  case (*Cons x w*)
  then show *?thesis*
  proof (*cases x*)
    case (*Pair a b*)
    then show *?thesis* **using** *Cons* **by** (*auto simp add*: *eq-wset-def fold-Some'*)
  qed
qed

lemma *list-split-eq-wset*:
  assumes *A*: *list-split xs ys*
  shows *eq-wset xs ys*
proof −
  have *H1*: *xs* ≠ [] ⟹ *list-all* (λ(*a*, *b*). *a* = *x*) *xs* ⟹ *eq-wset* ((*x*, *w*) # *xs*) [(*x*, *foldl* (+) *w* (*map snd xs*))] **for** *x* :: *'a* **and** *xs* :: (*'a* × (*'w* :: *ref-ab-semigroup-add*)) *list* **and** *w* :: *'w*
  proof (*induction xs arbitrary*: *w*)
    case *Nil*
    then show *?case*
      unfolding *eq-wset-def*
      by *simp*
  next
    case (*Cons x' xs' w*)

```
    then show ?case
      by (cases x'; cases xs'; simp add: eq-wset-trans[OF eq-wset-elem-comb])
  qed
  have H2: xs ≠ [] ⟹ list-all (λ(a, b). a = y) xs ⟹ eq-wset xs [(y, fold' (map
snd xs))] for xs :: ('a × ('w :: ref-ab-semigroup-add)) list and y :: 'a
  proof (induction xs)
    case Nil
    then show ?case by simp
  next
    case (Cons x xs')
    then show ?case
    proof (cases x)
      case (Pair a w)
      show ?thesis
      proof (cases xs')
        case Nil
        then show ?thesis using Cons Pair by simp
      next
        case (Cons x' xs'')
        have xs'-ne: xs' ≠ [] using Cons by simp
        have all-xs': list-all (λ(a, b). a = y) xs' using Cons.prems by simp
        have a-y: a = y using Cons.prems Pair by simp
        have eq-wset ((y, w) # xs') [(y, foldl (+) w (map snd xs'))]
          using H1[OF xs'-ne all-xs'] .
        then show ?thesis using Pair a-y by simp
      qed
    qed
  qed
  from A show ?thesis
  proof (induction rule: list-split.induct)
    case Base
    then show ?case by simp
  next
    case (Split xs'' ys xs xs' w y)
    have eq-tail: eq-wset xs'' ys using Split.IH .
    have eq-head: eq-wset xs' [(y, fold' (map snd xs'))]
      using H2 Split.hyps(4) Split.hyps(5) by blast
    have eq-wset (xs' @ xs'') ([(y, fold' (map snd xs'))] @ ys)
      using eq-head eq-tail by (rule eq-wset-append)
    then show ?case using Split.hyps(2) Split.hyps(3) by simp
  qed
qed


lemma list-split-app: list-split xs (ys @ ys') ⟹ ∃ xs' xs''. list-split xs' ys ∧ list-split
xs'' ys' ∧ xs = xs' @ xs''
proof (induction ys arbitrary: xs)
  case Nil
  then show ?case by auto
```

next
case (*Cons y yss*)
obtain *a w* **where** *y-def*: $y = (a, w)$ **by** (*cases y*)
from *Cons.prems* **have** *split-y*: *list-split xs* $((a, w) \# (yss \mathbin{@} ys'))$
unfolding *y-def* **by** *simp*
then obtain *xs1 xs2* **where**
xs-def: $xs = xs1 \mathbin{@} xs2$
and *split-tail*: *list-split xs2* $(yss \mathbin{@} ys')$
and *w-def*: $w = fold'\ (map\ snd\ xs1)$
and *xs1-ne*: $xs1 \neq []$
and *xs1-all*: *list-all* $(\lambda(a', b).\ a' = a)\ xs1$
by (*auto elim*: *list-split.cases*)
from *Cons.IH*[*OF split-tail*] **obtain** $xs'\ xs''$ **where**
split-l: *list-split* $xs'\ yss$
and *split-r*: *list-split* $xs''\ ys'$
and *xs2-def*: $xs2 = xs' \mathbin{@} xs''$
by *blast*
have *split-left*: *list-split* $(xs1 \mathbin{@} xs')\ ((a, w) \# yss)$
using *split-l w-def xs1-ne xs1-all* **by** (*intro Split*) *simp-all*
show *?case*
unfolding *y-def*
using *split-left split-r xs-def xs2-def* **by** *auto*
qed

lemma *list-split-trans*:
assumes *H*:*list-split xs ys*
and *H1*: *list-split ys zs*
shows *list-split xs zs*
proof −
have *H2'*: *list-all* $(\lambda(a, b).\ a = e)\ xs \Longrightarrow \forall\ e'.\ filter\ (\lambda(a', \text{-}).\ e' = a')\ xs = (if\ e = e'\ then\ xs\ else\ [])$ **for** $e :: 'a$ **and** $xs :: ('a \times 'w)\ list$
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs'*)
then show *?case* **by** (*cases x*) *auto*
qed
have *fold-upd*: *fold* $(\lambda(a, w).\ (+)\ (Some\ w))\ (if\ e = fst\ y\ then\ y \#\ filter\ (\lambda(a', \text{-}).\ e = a')\ ys\ else\ filter\ (\lambda(a', \text{-}).\ e = a')\ ys)\ (ws\ e) =$
fold $(\lambda(a, w).\ (+)\ (Some\ w))\ (filter\ (\lambda(a', \text{-}).\ e = a')\ ys)\ ((ws(fst\ y := Some\ (snd\ y) + ws\ (fst\ y)))\ e)$ **for** $e :: 'a$ **and** $ws :: 'a \Rightarrow ('w :: ref\text{-}ab\text{-}semigroup\text{-}add)\ option$ **and** $ys :: ('a \times 'w)\ list$ **and** $y :: ('a \times 'w)$
proof (*cases y*)
case (*Pair a w*)
then show *?thesis* **by** (*cases* $e = a$) *simp-all*
qed
have *list-all-eq-aux*: *list-all* $(\lambda(a, b).\ a = e)\ xs \Longrightarrow$
$\forall\, a.\ fold\ (\lambda(a, w).\ (+)\ (Some\ w))\ (if\ e = a\ then\ xs\ else\ [])\ None = fold\ (\lambda(a,$

```
w). (+) (Some w)) (filter (λ(a', -). a = a') ys) (w a) ⟹ list-all (λ(a, b). a = e)
ys for e :: 'a and xs :: ('a × ('w :: ref-ab-semigroup-add)) list and ys :: ('a × 'w)
list and w :: 'a ⇒ 'w option
  proof (induction ys arbitrary: w)
    case Nil
    then show ?case by (simp add: case-prod-beta)
  next
    case (Cons y ys')
    assume all-xs: list-all (λ(a, b). a = e) xs
    assume hall: ∀ a. fold (λ(a, w). (+) (Some w)) (if e = a then xs else []) None
=
                    fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a') (y # ys'))
(w a)
    have hall': ∀ a. fold (λ(a, w). (+) (Some w)) (if e = a then xs else []) None =
                    fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a') ys')
                        ((w(fst y := Some (snd y) + w (fst y))) a)
      using hall by (simp add: fold-upd[of - - - w] case-prod-beta)
    have ys'-all: list-all (λ(a, b). a = e) ys'
      using Cons.IH[of w(fst y := Some (snd y) + w (fst y))] all-xs hall' by fast
    have y-eq: fst y = e
    proof (rule ccontr)
      assume ne: fst y ≠ e
      have h-fy: fold (λ(a, w). (+) (Some w)) (if e = fst y then xs else []) None =
                fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). fst y = a') (y # ys'))
(w (fst y))
        using hall[THEN spec, of fst y] by simp
      have lhs: fold (λ(a, w). (+) (Some w)) (if e = fst y then xs else []) None =
None
        using ne by simp
      have rhs-pos: fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). fst y = a') (y #
ys')) (w (fst y)) ≠ None
        by (simp add: H2'[OF all-xs] H2' plus-option-case case-prod-beta fold-Some
            split: if-splits)
      then show False using lhs h-fy by simp
    qed
    show list-all (λ(a, b). a = e) (y # ys')
      using y-eq ys'-all by (cases y) auto
  qed
  have list-all-eq-wset: list-all (λ(a, b). a = e) xs ⟹ eq-wset xs ys ⟹ list-all
(λ(a, b). a = e) ys for e :: 'a and xs :: ('a × ('w :: ref-ab-semigroup-add)) list
and ys :: ('a × 'w) list
    unfolding eq-wset-def
    by(simp add: H2' list-all-eq-aux[of e xs ys λ-. None])
  have fold-Some-eq: fold (λ(a, w). (+) (Some w)) ys (Some wy) = fold (λ(a, w).
(+) (Some w)) xs (Some wx) ⟹
              fold (λx s. s + x) (map snd xs) wx = fold (λx s. s + x) (map snd ys)
wy for xs :: ('a × ('w :: ref-ab-semigroup-add)) list and ys :: ('a × 'w) list and
wx :: 'w and wy :: 'w
  proof (induction xs arbitrary: wx)
```

```
    case (Nil wx)
    then show ?case
    proof (induction ys arbitrary: wy)
      case Nil
      then show ?case by simp
    next
      case (Cons h t)
    then show ?case by (simp add: plus-option-case case-prod-beta ab-semigroup-add-class.add.commute)
    qed
  next
    case (Cons x xs' wx)
   then show ?case by (simp add: case-prod-beta ab-semigroup-add-class.add.commute)
  qed
  have fold-hd-tl-eq: xs ≠ [] ⟹
    fold (λ(a, w). (+) (Some w)) ys None = fold (λ(a, w). (+) (Some w)) xs None
⟹
    fold (λx s. s + x) (tl (map snd xs)) (hd (map snd xs)) = fold (λx s. s + x) (tl
(map snd ys)) (hd (map snd ys)) for xs :: ('a × ('w :: ref-ab-semigroup-add)) list
and ys :: ('a × 'w) list
  proof −
    assume ne: xs ≠ []
    assume feq: fold (λ(a, w). (+) (Some w)) ys None = fold (λ(a, w). (+) (Some
w)) xs None
    show fold (λx s. s + x) (tl (map snd xs)) (hd (map snd xs)) = fold (λx s. s +
x) (tl (map snd ys)) (hd (map snd ys))
    proof (cases xs)
      case Nil
      then show ?thesis using ne by simp
    next
      case (Cons x xs')
      then show ?thesis
      proof (cases ys)
        case Nil
          then show ?thesis using ‹xs = x # xs'› feq by (cases x; simp add:
fold-Some)
      next
        case (Cons y ys')
      then show ?thesis using ‹xs = x # xs'› feq by (auto simp add: case-prod-beta
intro: fold-Some-eq)
      qed
    qed
  qed
  have H2: xs ≠ [] ⟹
    list-all (λ(a, b). a = e) xs ⟹
    eq-wset ys xs ⟹ fold' (map snd xs) = fold' (map snd ys) ∧ ys ≠ [] ∧ list-all
(λ(a, b). a = e) ys for xs :: ('a × ('w :: ref-ab-semigroup-add)) list and ys :: ('a
× 'w) list and e :: 'a
  proof −
    assume ne: xs ≠ []
```

```
    assume all-xs: list-all (λ(a, b). a = e) xs
    assume eq: eq-wset ys xs
    have all-ys: list-all (λ(a, b). a = e) ys
      using list-all-eq-wset[OF all-xs, OF eq-wset-sym[OF eq]] .
    have ys-ne: ys ≠ []
    proof −
      obtain a w xs' where xs-eq: xs = (a, w) # xs'
        using ne by (cases xs) auto
      have a-eq: a = e
        using all-xs xs-eq by simp
      have xs-pos: sum-key xs e ≠ None
        by (simp add: xs-eq a-eq case-prod-beta fold-Some H2'[OF all-xs])
      have eq-e: sum-key ys e = sum-key xs e
        using eq unfolding eq-wset-def by simp
      show ?thesis
        using all-ys eq-e xs-pos by (auto simp add: fold-Some)
    qed
    have fold-eq: fold' (map snd xs) = fold' (map snd ys)
    proof −
      have xs-e: sum-key xs e = fold (λ(a, w). (+) (Some w)) xs None
        by (simp add: H2'[OF all-xs])
      have ys-e: sum-key ys e = fold (λ(a, w). (+) (Some w)) ys None
        by (simp add: H2'[OF all-ys])
      have key: fold (λ(a, w). (+) (Some w)) ys None = fold (λ(a, w). (+) (Some
w)) xs None
        using eq xs-e ys-e unfolding eq-wset-def by simp
      show ?thesis
        using fold-hd-tl-eq[OF ne, OF key] ne ys-ne by (simp add: foldl-conv-fold)
    qed
    show fold' (map snd xs) = fold' (map snd ys) ∧ ys ≠ [] ∧ list-all (λ(a, b). a =
e) ys
      using fold-eq ys-ne all-ys by simp
  qed
  show ?thesis
    using H1 H
  proof (induction ys zs arbitrary: xs rule: list-split.induct)
    case (Base xs)
    then show ?case by simp
  next
    case (Split xs'' ys' xs' zs1' w y xs)
    from Split.prems Split.hyps(2) have split-app: list-split xs (zs1' @ xs'')
      by simp
    from list-split-app[OF split-app] obtain xs-a xs-b where
      xs-split: xs = xs-a @ xs-b
      and split-a: list-split xs-a zs1'
      and split-b: list-split xs-b xs''
      by blast
    have eq-a: eq-wset xs-a zs1'
      using list-split-eq-wset[OF split-a] .
```

```
    have props-a: fold' (map snd zs1') = fold' (map snd xs-a) ∧ xs-a ≠ [] ∧ list-all
(λ(a, b). a = y) xs-a
      using H2[OF Split.hyps(4) Split.hyps(5) eq-a] .
    have xs-a-ne: xs-a ≠ [] using props-a by blast
    have xs-a-all: list-all (λ(a, b). a = y) xs-a using props-a by blast
    have w-xs-a: w = fold' (map snd xs-a)
      using Split.hyps(3) props-a by simp
    have split-b': list-split xs-b ys'
      using Split.IH[OF split-b] .
    show ?case
      unfolding xs-split
      by (intro list-split.Split[OF split-b' - w-xs-a xs-a-ne xs-a-all])
        simp
  qed
qed

lemma list-split'-length: list-split' xs ys ⟹ length xs = length ys
  by (induction xs arbitrary: ys) (auto elim: list-split'.cases)

lemma foldl-assoc: (⋀a b c. f (f a b) c = f a  (f b c)) ⟹ f z (foldl f y xs) = foldl
f (f z y) xs
  by (induction xs arbitrary: z y) auto

lemma list-split'-refl: list-split' (map (λx. [x]) xs) xs
  by (induction xs) (auto intro: list-split'.intros)

fun option-list :: ('a option) list ⇒ 'a list where
  option-list [] = [] |
  option-list (None # l) = option-list l |
  option-list (Some a # l) = a # option-list l

lemma option-list-eq-filter: filter ((≠) None) l1 = filter ((≠) None) l2 ⟹ option-list
l1 = option-list l2
proof (induction length l1 + length l2 arbitrary: l1 l2)
  fix l1 :: 'b option list
    and l2 :: 'b option list
  assume 0 = length (l1::'b option list) + length (l2::'b option list)
    and filter ((≠) None) l1 = filter ((≠) None) l2
  then show (option-list l1::'b list) = option-list l2
    by simp
next
  fix x :: nat
    and l1 :: 'b option list
    and l2 :: 'b option list
  assume ind: ⋀(l1 :: 'b option list) l2. x = length l1 + length l2 ⟹ filter ((≠)
None) l1 = filter ((≠) None) l2 ⟹ option-list l1 = option-list l2
    and len: Suc x = length (l1::'b option list) + length (l2::'b option list)
    and eq-l: filter ((≠) None) l1 = filter ((≠) None) l2
  consider l1 = [] ∨ l2 = [] | l1 ≠ [] ∧ l2 ≠ []
```

```
  by auto
then show (option-list l1::'b list) = option-list l2
proof (cases, goal-cases eq noteq)
  case eq
  then show ?case
  proof (cases l1 = [])
    case True
    show ?thesis
    proof (cases l2)
      case Nil
      then show ?thesis using True by simp
    next
      case (Cons h2 t2)
      then show ?thesis
      proof (cases h2)
        case None
        then show ?thesis using True Cons ind[of l1 t2] len eq-l by simp
      next
        case (Some v)
        then show ?thesis using True Cons eq-l by simp
      qed
    qed
  next
    case False
    with eq have l2 = [] by simp
    show ?thesis
    proof (cases l1)
      case Nil
      then show ?thesis using False by simp
    next
      case (Cons h1 t1)
      then show ?thesis
      proof (cases h1)
        case None
        then show ?thesis using ‹l2 = []› Cons ind[of t1 l2] len eq-l by simp
      next
        case (Some v)
        then show ?thesis using False ‹l2 = []› Cons eq-l by simp
      qed
    qed
  qed
next
  case noteq
  then show ?case
  proof (cases l1)
    case Nil
    then show ?thesis using noteq by simp
  next
    case (Cons h1 t1)
```

```
      show ?thesis
      proof (cases l2)
        case Nil
        then show ?thesis using noteq by simp
      next
        case (Cons h2 t2)
        then show ?thesis
        proof (cases h1)
          case None
          then show ?thesis using ‹l1 = h1 # t1› Cons ind[of t1 l2] len eq-l by
simp
        next
          case (Some v1)
          then show ?thesis
          proof (cases h2)
            case None
            then show ?thesis using ‹l1 = h1 # t1› Cons ‹h1 = Some v1› ind[of
l1 t2] len eq-l by simp
          next
            case (Some v2)
            then show ?thesis using ‹l1 = h1 # t1› Cons ‹h1 = Some v1› ind[of
t1 None # t2] len eq-l by simp
          qed
        qed
      qed
    qed
  qed
qed


lemma fold-option-not-none: Some a = fold' l ⟹ l ≠ [] ⟹ (option-list l) ≠ []
proof (induction l arbitrary: a)
  case Nil
  then show ?case by simp
next
  case (Cons h t)
  then show ?case by (cases h; cases t) auto
qed


lemma fold-option: Some a = fold' l ⟹ l ≠ [] ⟹ a = fold' (option-list l)
proof −
  have H: Some x = foldl (+) (Some x') l ⟹ x = foldl (+) x' (option-list l)
    for x x' and l :: 'a option list
  proof (induction l arbitrary: x x')
    case Nil
    then show ?case by simp
  next
```

```
    case (Cons a l)
   then show ?case by (cases a) (auto simp add: plus-option-def split: option.split)
  qed
  show Some a = fold' l ⟹ l ≠ [] ⟹ a = fold' (option-list l)
  proof (induction l arbitrary: a)
    case Nil
    then show ?case by simp
  next
    case (Cons h t)
    then show ?case
    proof (cases h)
      case None
      then show ?thesis using Cons by (cases t) auto
    next
      case (Some a')
      then show ?thesis using H[of a a' t] Cons by simp
    qed
  qed
qed

fun create-split :: ('a × 'w) list ⇒ ('a ⇒ (('w option) list) list) ⇒ ('a × 'w) list
where
  create-split [] als = [] |
   create-split ((a,-) # l) als =  map (λx. (a,x)) (option-list (hd (als a))) @
(create-split l (als(a := tl(als a))))

lemma list-split'-exist:
  ((xs ≠ [] ∧ ys ≠ []) ⟶ fold' (xs :: (('w :: ref-ab-semigroup-add) option) list) =
fold' ys) ⟹
    ((xs = []) = (ys = [])) ⟹
  (∃ zs zs'. list-split' zs xs ∧ list-split' zs' ys ∧ (∀ n m. n < length xs ⟶ m < length
ys ⟶ zs ! n ! m = zs' ! m ! n) ∧
    list-all (λl. length l = length ys) zs ∧ list-all (λl. length l = length xs) zs')
proof (induction length xs + length ys arbitrary: xs ys rule: nat-less-induct)
  fix xs :: ('w option) list
    and ys :: ('w option) list
  assume ind: ∀ m<length (xs :: (('w :: ref-ab-semigroup-add) option) list) + length
ys.
          ∀ (x :: (('w :: ref-ab-semigroup-add) option) list) ls'.
             m = length x + length ls' ⟶
             (x ≠ [] ∧ ls' ≠ [] ⟶ fold' x = fold' ls') ⟶
             (x = []) = (ls' = []) ⟶
             (∃ zs zs'.
                  list-split' zs x ∧ list-split' zs' ls' ∧ (∀ n m. n < length x ⟶ m <
length ls' ⟶ zs ! n ! m = zs' ! m ! n) ∧ list-all (λl. length l = length ls') zs ∧
list-all (λl. length l = length x) zs')
    and eq-app: (xs ≠ [] ∧ ys ≠ []) ⟶ fold' (xs :: (('w :: ref-ab-semigroup-add)
option) list) = fold' ys
    and eq-nil: (xs = []) = (ys = [])
```

have *ind*: $\bigwedge(x :: (('w :: \textit{ref-ab-semigroup-add})\ \textit{option})\ \textit{list})\ ls'.$
$\textit{length}\ x + \textit{length}\ ls' < \textit{length}\ xs + \textit{length}\ ys \Longrightarrow$
$(x \neq [] \wedge ls' \neq [] \longrightarrow \textit{fold}'\ x = \textit{fold}'\ ls') \Longrightarrow$
$(x = []) = (ls' = []) \Longrightarrow$
$(\exists\, zs\ zs'.$
$\textit{list-split}'\ zs\ x \wedge \textit{list-split}'\ zs'\ ls' \wedge (\forall\, n\ m.\ n < \textit{length}\ x \longrightarrow m < \textit{length}\ ls' \longrightarrow zs\ !\ n\ !\ m = zs'\ !\ m\ !\ n) \wedge \textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ ls')\ zs \wedge \textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ x)\ zs')$
using *ind* **by** *auto*
have *gt3-cases*: $\bigwedge(xs :: 'w\ \textit{option list})\ ys.\ ((xs = []) = (ys = [])) \Longrightarrow$
$((xs \neq [] \wedge ys \neq []) \longrightarrow \textit{fold}'\ (xs :: (('w :: \textit{ref-ab-semigroup-add})\ \textit{option})\ \textit{list}) = \textit{fold}'\ ys) \Longrightarrow$
$(\bigwedge(x :: (('w :: \textit{ref-ab-semigroup-add})\ \textit{option})\ \textit{list})\ ls'.$
$\textit{length}\ x + \textit{length}\ ls' < \textit{length}\ xs + \textit{length}\ ys \Longrightarrow$
$(x \neq [] \wedge ls' \neq [] \longrightarrow \textit{fold}'\ x = \textit{fold}'\ ls') \Longrightarrow$
$(x = []) = (ls' = []) \Longrightarrow$
$(\exists\, zs\ zs'.$
$\textit{list-split}'\ zs\ x \wedge \textit{list-split}'\ zs'\ ls' \wedge (\forall\, n\ m.\ n < \textit{length}\ x \longrightarrow m < \textit{length}\ ls' \longrightarrow zs\ !\ n\ !\ m = zs'\ !\ m\ !\ n) \wedge \textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ ls')\ zs \wedge \textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ x)\ zs')) \Longrightarrow$
$2 < \textit{length}\ xs \Longrightarrow \exists\, zs\ zs'.$
$\textit{list-split}'\ zs\ xs \wedge$
$\textit{list-split}'\ zs'\ ys \wedge$
$(\forall\, n\ m.\ n < \textit{length}\ xs \longrightarrow m < \textit{length}\ ys \longrightarrow zs\ !\ n\ !\ m = zs'\ !\ m\ !\ n) \wedge$
$\textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ ys)\ zs \wedge \textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ xs)\ zs'$
proof −
fix *xs* :: *'w option list* **and** *ys* :: *'w option list*
assume *len*: $2 < \textit{length}\ xs$
and *eq-nil*: $(xs = []) = (ys = [])$
and *eq-app*: $(xs \neq [] \wedge ys \neq []) \longrightarrow \textit{fold}'\ (xs :: (('w :: \textit{ref-ab-semigroup-add})\ \textit{option})\ \textit{list}) = \textit{fold}'\ ys$
and *ind*: $\bigwedge(x :: (('w :: \textit{ref-ab-semigroup-add})\ \textit{option})\ \textit{list})\ ls'.$
$\textit{length}\ x + \textit{length}\ ls' < \textit{length}\ xs + \textit{length}\ ys \Longrightarrow$
$(x \neq [] \wedge ls' \neq [] \longrightarrow \textit{fold}'\ x = \textit{fold}'\ ls') \Longrightarrow$
$(x = []) = (ls' = []) \Longrightarrow$
$(\exists\, zs\ zs'.$
$\textit{list-split}'\ zs\ x \wedge \textit{list-split}'\ zs'\ ls' \wedge (\forall\, n\ m.\ n < \textit{length}\ x \longrightarrow m < \textit{length}\ ls' \longrightarrow zs\ !\ n\ !\ m = zs'\ !\ m\ !\ n) \wedge \textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ ls')\ zs \wedge \textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ x)\ zs')$
obtain *x x' xs'* **where** *xs-def*: $xs = x \# x' \# xs'$ **and** *xs'-nil*: $xs' \neq []$
using *len*
by (*metis One-nat-def Suc-1 Suc-eq-plus1 length-0-conv length-Cons less-nat-zero-code list.exhaust not-add-less1 verit-comp-simplify1*(1))
have $\exists\ zs\ zs'.\ \textit{list-split}'\ zs\ ((x + x') \# xs') \wedge$
$\textit{list-split}'\ zs'\ ys \wedge$
$(\forall\, n\ m.\ n < \textit{length}\ ((x + x') \# xs') \longrightarrow m < \textit{length}\ ys \longrightarrow zs\ !\ n\ !\ m = zs'\ !\ m\ !\ n) \wedge$
$\textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ ys)\ zs \wedge \textit{list-all}\ (\lambda l.\ \textit{length}\ l = \textit{length}\ ((x + x') \# xs'))\ zs'$

using *ind*[*of* $(x + x')$ # xs' *ys*] *eq-app eq-nil*
unfolding *xs-def*
by *auto*
obtain *zs zs'* **where** *ind1*: *list-split'* *zs* $((x + x')$ # $xs') \wedge$
list-split' zs' *ys* $\wedge$
$(\forall\, n\ m.\ n < length\ ((x + x')\ \#\ xs') \longrightarrow m < length\ ys \longrightarrow zs\ !\ n\ !\ m = zs'\ !\ m\ !\ n) \wedge$
list-all $(\lambda l.\ length\ l = length\ ys)$ *zs* $\wedge$ *list-all* $(\lambda l.\ length\ l = length\ ((x + x')\ \#\ xs'))$ zs'
using *ind*[*of* $(x + x')$ # xs' *ys*] *eq-app eq-nil*
unfolding *xs-def*
by *auto*
have $\exists$ *zsa zs'*.
list-split' *zsa* $[x, x'] \wedge$
list-split' zs' (*hd zs*) $\wedge$
$(\forall\, n\ m.\ n < length\ [x, x'] \longrightarrow m < length\ (hd\ zs) \longrightarrow zsa\ !\ n\ !\ m = zs'\ !\ m\ !\ n) \wedge$ *list-all* $(\lambda l.\ length\ l = length\ (hd\ zs))$ *zsa* $\wedge$ *list-all* $(\lambda l.\ length\ l = length\ [x, x'])$ zs'
proof (*rule ind*[*of* $[x, x']$ *hd zs*])
show *length* $[x, x']$ + *length* (*hd zs*) < *length xs* + *length ys*
using *len eq-nil ind1*
by (*cases zs*) (*auto elim*: *list-split'.cases simp add*: *less-eq-Suc-le*)
show $[x, x'] \neq [] \wedge$ *hd zs* $\neq []$ $\longrightarrow$ *fold'* $[x, x']$ = *fold'* (*hd zs*)
using *len eq-nil ind1*
by (*auto elim*: *list-split'.cases simp add*: *ab-semigroup-add-class.add.commute*)
show $([x, x'] = [])$ = (*hd zs* = [])
using *len eq-nil ind1*
by (*auto elim*: *list-split'.cases simp add*: *ab-semigroup-add-class.add.commute*)
qed
then obtain *zs1 zs1'* **where** *ind2*: *list-split'* *zs1* $[x, x'] \wedge$ *list-split'* *zs1'* (*hd zs*) $\wedge$
$(\forall\, n\ m.\ n < length\ [x, x'] \longrightarrow m < length\ (hd\ zs) \longrightarrow zs1\ !\ n\ !\ m = zs1'\ !\ m\ !\ n) \wedge$ *list-all* $(\lambda l.\ length\ l = length\ (hd\ zs))$ *zs1* $\wedge$ *list-all* $(\lambda l.\ length\ l = length\ [x, x'])$ *zs1'*
by *blast*
have *H1*: *list-split'* (*zs1* ! *0* # *zs1* ! *1* # *tl zs*) *xs*
proof (*cases zs1*)
case *Nil*
then show *?thesis* **using** *ind1 ind2* **unfolding** *xs-def* **by** (*auto elim*: *list-split'.cases*)
next
case (*Cons e zss1*)
then show *?thesis* **using** *ind1 ind2* **unfolding** *xs-def*
by (*cases zss1*) (*auto intro*!: *list-split'.intros elim*: *list-split'.cases*)
qed
have $\bigwedge m\ n.\ m < length\ ys \longrightarrow zs\ !\ 0\ !\ m = zs'\ !\ m\ !\ 0$ *list-split'* zs' *ys list-split'* *zs1'* (*hd zs*) *list-all* $(\lambda l.\ length\ l = length\ [x, x'])$ *zs1'* *list-all* $(\lambda l.\ length\ l = length\ ys)$ *zs*
using *ind1 ind2* **by** *auto*

also have *len-zs1'*: *length zs1' = length ys*
using *ind1 ind2 list-split'-length*
by (*metis* (*mono-tags*, *lifting*) *hd-conv-nth length-greater-0-conv list.distinct*(*1*) *list-all-length*)
moreover have *ys ≠ [] ⟹ zs ≠ []*
using *ind1*
by (*auto elim*: *list-split'.cases*)
ultimately have *H2*: *list-split' (map2 (λl l'. (l' ! 0) # (l' ! 1) # (tl l)) zs' zs1') ys*
proof (*induction ys arbitrary*: *zs zs' zs1'*)
case *Nil*
then show *?case*
by(*auto intro*: *list-split'.intros dest*: *list-split'.cases*)
next
fix *y* :: *'w option*
and *ys* :: *'w option list*
and *zs* :: *'w option list list*
and *zs'* :: *'w option list list*
and *zs1'* :: *'w option list list*
assume *ind*: *⋀zs zs' zs1'. (⋀m. m < length ys ⟶ zs ! 0 ! m = zs' ! m ! 0) ⟹ list-split' zs' ys ⟹ list-split' zs1' (hd zs) ⟹ list-all (λl. length l = length [x, x']) zs1' ⟹ list-all (λl. length l = length ys) zs ⟹ length zs1' = length ys ⟹ (ys ≠ [] ⟹ zs ≠ []) ⟹ list-split' (map2 (λx y. y ! 0 # y ! 1 # tl x) zs' zs1') ys*
and *trans*: *⋀m. m < length (y # ys) ⟶ (zs ! 0 ! m::'w option) = zs' ! m ! 0*
and *zs'-ys*: *list-split' zs' (y # ys)*
and *zs1'-zs*: *list-split' zs1' (hd zs::'w option list)*
and *len*: *list-all (λl. length (l::'w option list) = length [x, x']) zs1'*
and *len-zs*: *list-all (λl. length l = length (y # ys)) zs*
and *len'*: *length zs1' = length (y # ys)*
and *zs-Nil*: *(y # ys) ≠ [] ⟹ zs ≠ []*
have *trans'*: *zs ! 0 ! 0 = zs' ! 0 ! 0*
using *trans* **by** *auto*
have *zs-Nil*: *zs ≠ []*
using *zs-Nil* **by** *simp*
obtain *z zss'* **where** *zs'-def*: *zs' = z # zss'* **and** *y-fold*: *y = fold' z* **and** *zss'-ys*: *list-split' zss' ys* **and** *z-Nil*: *z ≠ []*
using *zs'-ys*
by(*auto elim*: *list-split'.cases*)
obtain *ze zl* **where** *z-app*: *z = ze # zl*
using *z-Nil*
by (*meson list.exhaust*)
have *ze-def*: *ze = zs ! 0 ! 0*
using *trans'*
unfolding *zs'-def z-app*
by *simp*
obtain *zse zsl* **where** *zs-def*: *zs = zse # zsl*
using *zs-Nil*
using *list.exhaust* **by** *auto*

```
      have exist-zs1': ∃ zs1'e1 zs1'e2 zs1'l. zs1' = [zs1'e1, zs1'e2] # zs1'l ∧
zs1'e1 + zs1'e2 = ze
    proof (cases rule: list-split'.cases[OF zs1'-zs])
      case 1
      then show ?thesis using len' zs1'-zs by auto
    next
      case (2 xss ys' y xs)
      then show ?thesis
        using len ze-def zs-def
        by (cases xs; cases tl xs) auto
    qed
    then obtain zs1'e1 zs1'e2 zs1'l where zs1'-def: zs1' = [zs1'e1, zs1'e2] #
zs1'l ∧ zs1'e1 + zs1'e2 = ze
      by auto
    show list-split' (map2 (λx y. y ! 0 # y ! 1 # tl x) zs' zs1') (y # ys)
    proof −
      have H: zs1' = [zs1'e1, zs1'e2] # zs1'l using zs1'-def by simp
      have goal: list-split' ((zs1'e1 # zs1'e2 # tl z) # map2 (λx y. y ! 0 # y !
Suc 0 # tl x) zss' zs1'l) (y # ys)
      proof (rule list-split'.intros)
        show list-split' (map2 (λx y. y ! 0 # y ! Suc 0 # tl x) zss' zs1'l) ys
          unfolding One-nat-def[symmetric]
        proof (rule ind[of map tl zs])
          show ⋀m. m < length ys ⟶ map tl zs ! 0 ! m = zss' ! m ! 0
          proof −
            fix m show m < length ys ⟶ map tl zs ! 0 ! m = zss' ! m ! 0
              using trans[of Suc m] zs-def zs'-def len-zs nth-tl[of m zse] by simp
          qed
        next
          show list-split' zss' ys
            using zss'-ys by simp
        next
          show list-split' zs1'l (hd (map tl zs))
            using zs1'-zs zs1'-def zs-def
            by (auto elim: list-split'.cases)
        next
          show list-all (λl. length l = length [x, x']) zs1'l
            using len zs1'-def by simp
        next
          show list-all (λl. length l = length ys) (map tl zs)
            using len-zs by (induction zs; simp)
        next
          show length zs1'l = length ys
            using len' zs1'-def by simp
        next
          show ys ≠ [] ⟹ map tl zs ≠ []
            unfolding zs-def by simp
        qed
      next
```

```
      show y = fold' (zs1'e1 # zs1'e2 # tl z)
        using y-fold z-app zs1'-def by simp
    next
      show zs1'e1 # zs1'e2 # tl z ≠ [] by simp
    qed
    show ?thesis
      using goal H
      unfolding zs'-def ze-def trans'
      by simp
  qed
qed
have Heq: length zs' = length zs1'
  using ind1 ind2
  by (cases zs) (auto dest!: list-split'-length)
have length zs' = length zs1' ⟹ map2 (λx y. y ! 0 # y ! Suc 0 # tl x) zs'
zs1' ! m ! 0 = zs1' ! m ! 0 for m
proof (induction m arbitrary: zs' zs1')
  case 0
  then show ?case by(cases zs'; cases zs1'; auto)
next
  case (Suc m)
  then show ?case by(cases zs'; cases zs1'; auto)
qed
then have H3-1: map2 (λx y. y ! 0 # y ! Suc 0 # tl x) zs' zs1' ! m ! 0 =
zs1' ! m ! 0 for m
  using Heq
  by blast
have length zs' = length zs1' ⟹ map2 (λx y. y ! 0 # y ! Suc 0 # tl x) zs'
zs1' ! m ! Suc 0 = zs1' ! m ! Suc 0 for m
proof (induction m arbitrary: zs' zs1')
  case 0
  then show ?case by(cases zs'; cases zs1'; auto)
next
  case (Suc m)
  then show ?case by(cases zs'; cases zs1'; auto)
qed
then have H3-2: ⋀ m. map2 (λx y. y ! 0 # y ! Suc 0 # tl x) zs' zs1' ! m !
Suc 0 = zs1' ! m ! Suc 0
  using Heq
  by blast
have H3-3: n ≥ 2 ⟹ m < length zs' ⟹ length zs' = length zs1' ⟹ n ≤
length (zs' ! m) ⟹ map2 (λx y. y ! 0 # y ! Suc 0 # tl x) zs' zs1' ! m ! n = zs'
! m ! (n − 1) for n m
proof (induction m arbitrary: zs' zs1')
  case 0
  then show ?case
    by (cases zs'; cases zs1') (auto simp: nth-tl Suc-diff-Suc)
next
  case (Suc m)
```

```
    then show ?case
      by (cases zs'; cases zs1'; auto)
  qed
  have H3-help: n < length xs ⟹ m < length zs ⟹ list-all (λl. length l = Suc
(length xs)) zs ⟹ Suc (Suc n) ≤ length (zs ! m)
    for n m and xs :: 'a list and zs :: 'b list list
  proof (induction zs arbitrary: n m xs)
    case (Cons a zs)
    then show ?case
      by(cases m; simp)
  qed simp
  have H3: ∀ n m. n < length xs ⟶ m < length ys ⟶ (zs1 ! 0 # zs1 ! 1 # tl
zs) ! n ! m = map2 (λzs' y. y ! 0 # y ! 1 # tl zs') zs' zs1' ! m ! n
  proof (intro allI impI)
    fix n m
    assume hn: n < length xs and hm: m < length ys
    show (zs1 ! 0 # zs1 ! 1 # tl zs) ! n ! m = map2 (λzs' y. y ! 0 # y ! 1 # tl
zs') zs' zs1' ! m ! n
    proof (cases n)
      case 0
      have zs1 ! 0 ! m = zs1' ! m ! 0
        using ind2 len-zs1' hm by(auto dest: list-split'-length)
      then show ?thesis using ‹n = 0› by (simp add: H3-1[symmetric])
    next
      case (Suc n')
      show ?thesis
      proof (cases n')
        case 0
        have zs1 ! 1 ! m = zs1' ! m ! 1
          using ind2 len-zs1' hm by(auto dest: list-split'-length)
            then show ?thesis using ‹n = Suc n'› ‹n' = 0› by (simp add:
H3-2[symmetric])
      next
        case (Suc n'')
        have hn2: Suc (Suc n'') ≤ 2 by simp
        have hm2: m < length zs' using ind1 hm by (auto dest: list-split'-length)
        have hlen: length zs' = length zs1' using Heq by simp
        have hlen2: Suc (Suc n'') ≤ length (zs' ! m)
          using ind1 xs-def ‹n = Suc n'› ‹n' = Suc n''› hn hm2
          by(fastforce intro: H3-help dest!: list-split'-length)
        have step1: map2 (λzs' y. y ! 0 # y ! 1 # tl zs') zs' zs1' ! m ! n = zs' !
m ! (n − 1)
          using H3-3[OF hn2 hm2 hlen hlen2] ‹n = Suc n'› ‹n' = Suc n''› by
simp
        have ntl: tl zs ! n'' = zs ! Suc n''
          using ind1 xs-def ‹n = Suc n'› ‹n' = Suc n''› hn
          by(fastforce intro: nth-tl dest: list-split'-length)
        have step2: (zs1 ! 0 # zs1 ! 1 # tl zs) ! n ! m = zs' ! m ! (n − 1)
          using ind1 hm ntl xs-def ‹n = Suc n'› ‹n' = Suc n''› hn
```

```
          by(fastforce dest: list-split'-length)
        with step1 show ?thesis by simp
      qed
    qed
  qed
  have H4: list-all (λl. length l = length ys) (zs1 ! 0 # zs1 ! 1 # tl zs)
    using ind1 ind2
    unfolding xs-def
  proof (cases zs1)
    case Nil
    then show ?thesis using ind2 by (auto elim: list-split'.cases)
  next
    case (Cons e zss1)
   then show ?thesis using ind1 ind2 by (cases zss1) (auto intro!: list-split'.intros
elim: list-split'.cases)
  qed
  have list-all (λl. length l = length ((x + x') # xs')) zs' length zs' = length zs1'
    using ind1 Heq by auto
   then have H5: list-all (λl. length l = length xs) (map2 (λzs' y. y ! 0 # y !
Suc 0 # tl zs') zs' zs1')
    unfolding xs-def
  proof (induction zs' arbitrary: zs1')
    case Nil then show ?case by simp
  next
    case (Cons z zs') then show ?case by (cases zs1'; simp add: xs-def)
  qed
  show ∃ zs zs'. list-split' zs xs ∧ list-split' zs' ys ∧
                (∀ n m. n < length xs ⟶ m < length ys ⟶ zs ! n ! m = zs' ! m
! n) ∧
                list-all (λl. length l = length ys) zs ∧ list-all (λl. length l = length
xs) zs'
    using H1 H2 H3 H4 H5
    by (intro exI[where x = zs1 ! 0 # zs1 ! 1 # tl zs]
        exI[where x = map2 (λl l'. (l' ! 0) # (l' ! 1) # (tl l)) zs' zs1']) simp
 qed
 have list-all-len-1: ⋀ys. list-all (λl. length l = Suc 0) (map (λy. [y]) ys)
   by (simp add: list-all-length)
 consider length xs ≤ 1 | length ys ≤ 1 | length xs = 2 ∧ length ys = 2 | length
xs > 2 | length ys > 2
   by linarith
 then show ∃ zs zs'. list-split' zs xs ∧ list-split' zs' ys ∧ (∀ n m. n < length xs
⟶ m < length ys ⟶ zs ! n ! m = zs' ! m ! n) ∧
      list-all (λl. length l = length ys) zs ∧ list-all (λl. length l = length xs) zs'
 proof (cases, goal-cases leq1 leq1' eq2 gt3 gt3')
   case leq1
   then show ?case
   proof (cases xs)
     case Nil
     then show ?thesis
```

```
      using eq-nil eq-app
        by (intro exI[where x = []] exI[where x = []]) (cases ys; auto intro:
list-split'.intros)
    next
      case (Cons x xs')
      then show ?thesis
        using eq-nil eq-app list-all-len-1 leq1
        by (intro exI[where x = [ys]] exI[where x = map (λy. [y]) ys])
          (auto intro: list-split'.intros list-split'-refl simp add: list-all-length)
    qed
  next
    case leq1'
    then show ?case
    proof (cases ys)
      case Nil
      then show ?thesis
        using eq-nil eq-app
        by (intro exI[where x = []]) (cases xs; auto intro: list-split'.intros)
    next
      case (Cons y ys')
      then show ?thesis
        using eq-nil eq-app list-all-len-1 leq1'
        by (intro exI[where x = map (λx. [x]) xs] exI[where x = [xs]])
          (auto intro: list-split'.intros list-split'-refl simp add: list-all-length)
    qed
  next
    case eq2
    then show ?case
    proof (cases xs; cases ys)
      fix x xs' y ys'
      assume xs-def2: xs = x # xs' and ys-def2: ys = y # ys'
      with eq2 have length xs' = 1 length ys' = 1 by auto
      then obtain x' y' where xs'-def: xs' = [x'] and ys'-def: ys' = [y']
        by (cases xs'; cases ys'; auto)
      have heq: x + x' = y + y'
        using eq-app unfolding xs-def2 ys-def2 xs'-def ys'-def by simp
      obtain e11 e12 e21 e22 where
        ref: Some x = e11 + e12 Some x' = e21 + e22
        Some y = e11 + e21 Some y' = e12 + e22
        using refinable[OF heq] by blast
      let ?zs = [case (e11, e12) of (Some e11', Some e12') ⇒ [e11', e12'] | (Some
e11', None) ⇒ [e11', None] | (None, Some e12') ⇒ [None, e12'],
                 case (e21, e22) of (Some e21', Some e22') ⇒ [e21', e22'] | (Some
e21', None) ⇒ [e21', None] | (None, Some e22') ⇒ [None, e22']]
      let ?zs' = [case (e11, e21) of (Some e11', Some e21') ⇒ [e11', e21'] | (Some
e11', None) ⇒ [e11', None] | (None, Some e21') ⇒ [None, e21'],
                 case (e12, e22) of (Some e12', Some e22') ⇒ [e12', e22'] | (Some
e12', None) ⇒ [e12', None] | (None, Some e22') ⇒ [None, e22']]
      have zs-split: list-split' ?zs xs
```

```
      using ref unfolding xs-def2 xs'-def
      by (cases e11; cases e12; cases e21; cases e22)
     (auto intro!: list-split'.intros simp add: ab-semigroup-add-class.add.commute)
    have zs'-split: list-split' ?zs' ys
      using ref unfolding ys-def2 ys'-def
      by (cases e11; cases e12; cases e21; cases e22)
     (auto intro!: list-split'.intros simp add: ab-semigroup-add-class.add.commute)
    have cross: ∀ n m. n < length xs ⟶ m < length ys ⟶ ?zs ! n ! m = ?zs' !
m ! n
      using ref unfolding xs-def2 xs'-def ys-def2 ys'-def
      by (cases e11; cases e12; cases e21; cases e22)
      (auto intro!: list-split'.intros simp add: ab-semigroup-add-class.add.commute
          | metis less-Suc0 less-antisym nth-Cons-0 nth-Cons-Suc)+
    have len-zs: list-all (λl. length l = length ys) ?zs
      using ref unfolding ys-def2 ys'-def
      by (cases e11; cases e12; cases e21; cases e22) simp-all
    have len-zs': list-all (λl. length l = length xs) ?zs'
      using ref unfolding xs-def2 xs'-def
      by (cases e11; cases e12; cases e21; cases e22) simp-all
    show ?thesis
      by (intro exI[where x = ?zs] exI[where x = ?zs'])
       (use zs-split zs'-split cross len-zs len-zs' in simp)
   qed (auto simp: eq2)
  next
   case gt3
   assume len: 2 < length xs
   show ?case
    using gt3-cases len ind eq-app eq-nil
    by presburger
  next
   case gt3'
   assume len: 2 < length ys
   have (ys = []) = (xs = [])
    using eq-nil by simp
   moreover have ys ≠ [] ∧ xs ≠ [] ⟶ fold' ys = fold' xs
    using eq-app by simp
   moreover have length x + length ls' < length ys + length xs ⟹
     x ≠ [] ∧ ls' ≠ [] ⟶ fold' x = fold' ls' ⟹
     (x = []) = (ls' = []) ⟹
     ∃ zs zs'.
       list-split' zs x ∧
       list-split' zs' ls' ∧
       (∀ n m. n < length x ⟶ m < length ls' ⟶ zs ! n ! m = zs' ! m ! n) ∧
       list-all (λl. length l = length ls') zs ∧ list-all (λl. length l = length x) zs'
for x :: 'w option list and ls'
    using ind[of x ls']
    by auto
   ultimately show ?case
    using len gt3-cases[of ys xs]
```

```
    by(simp, metis)
  qed
qed


lemma create-split-count:
  length (zs a) = length (filter (λ(x, -). a = x) xs) ⟹
     count (mset (concat (zs a))) (Some w) = count (mset (create-split xs zs)) (a,
w)
  for zs :: 'a ⇒ ('w :: ref-ab-semigroup-add) option list list
  and a :: 'a and xs :: ('a × 'w) list and w :: 'w
proof (induction xs arbitrary: zs)
    case Nil
    then show ?case by simp
  next
    case (Cons x xs)
    obtain x1 x2 where x-def: x = (x1, x2)
      by fastforce
    show ?case
    proof (cases a = x1)
      case True
      obtain z zsa where zs-def: zs x1 = z # zsa
        using Cons.prems True
        unfolding x-def
        by(cases zs a; simp)
      have step1: count (mset z) (Some w) = count (image-mset (Pair x1) (mset
(option-list z))) (x1, w)
      proof (induction z)
        case (Cons z1 z2) then show ?case by (cases z1; simp)
      qed simp
      have step2: count (mset (concat zsa)) (Some w) = count (mset (create-split
xs (zs(x1 := zsa)))) (x1, w)
        using Cons.IH[of zs(x1 := zsa)] Cons.prems
        unfolding True x-def zs-def
        by (simp add: fun-upd-def)
      show ?thesis
        using step1 step2 zs-def
        unfolding x-def True
        by simp
    next
      case neq: False
      have cnt0: ⋀m. count (image-mset (Pair x1) m) (a, w) = 0
        using neq prod.inject by fastforce
      have len: length (zs a) = length (filter (λ(x, -). a = x) xs)
        using Cons.prems neq
        unfolding x-def
        by simp
      show ?thesis
        using Cons.IH[of (zs(x1 := tl (zs x1)))] neq cnt0 len
```

```
      unfolding x-def
      by (simp add: fun-upd-def)
    qed
  qed

lemma list-split-exist:
  assumes wset-eq: eq-wset xs ys
  shows ∃ zs zs'. list-split zs xs ∧ list-split zs' ys ∧ mset zs = mset zs'
proof −
  have foldl-eq: Some x + foldl (+) (Some b) (map (λx. Some (snd x)) zs) = foldl
(+) (Some x + Some b) (map (λx. Some (snd x)) zs) for x b and zs :: ('a × 'w ::
ab-semigroup-add) list
   by (induction zs arbitrary: x b) (auto simp add: plus-option-def add.left-commute
add.assoc)
  have filter-eq: filter (λ(x, -). a = x) xs = [] = (weight (abs-wset xs) a = None)
for xs :: ('a × 'w :: ref-ab-semigroup-add) list and a
  proof (induction xs)
    case (Cons x xs)
    then show ?case
      by (cases x) (auto simp add: weight.abs-eq plus-option-def fold-Some)
  qed (simp add: wempty-def weight.abs-eq)
  have weight-wset-eq: filter (λ(x, -). a = x) xs ≠ [] ⟹ weight (abs-wset xs) a =
fold' (map (λx. Some (snd x)) (filter (λ(x, -). a = x) xs))
    for xs :: ('a × 'w :: ref-ab-semigroup-add) list and a
  proof −
      assume hne: filter (λ(x, -). a = x) xs ≠ []
      obtain b list where hfl: filter (λ(x, -). a = x) xs = b # list
        using hne by (cases filter (λ(x, -). a = x) xs) auto
       have step: weight (abs-wset xs) a = fold' (map (λx. Some (snd x)) (filter
(λ(x, -). a = x) xs))
      proof −
        obtain b1 b2 where b-def: b = (b1, b2) by fastforce
        have rhs-eq: fold' (map (λx. Some (snd x)) (filter (λ(x, -). a = x) xs)) =
                    foldl (+) (Some b2) (map (λx. Some (snd x)) list)
          unfolding hfl b-def by simp
        have lhs-eq: weight (abs-wset xs) a =
                    fold (λ(-, w) y. Some w + y) (filter (λ(x, -). a = x) xs) None
          by (simp add: weight.abs-eq)
        have conn: fold (λ(-, w) y. Some w + y) ((b1, b2) # list) None =
                  foldl (+) (Some b2) (map (λx. Some (snd x)) list)
        by (induction list arbitrary: b2) (auto simp: add.commute add.left-commute)
        show ?thesis
          unfolding lhs-eq hfl b-def rhs-eq
          using conn by simp
      qed
      then show ?thesis by assumption
    qed
  have ind-help: ∀ n<length zs. ∀ m<length zs'. zs ! n ! m = zs' ! m ! n ⟹
    list-all (λl. length l = length zs') zs ⟹ list-all (λl. length l = length zs) zs'
```

$\Longrightarrow$ *mset* (*concat zs*) = *mset* (*concat zs'*)
for *zs* :: ((*'w option*) *list*) *list* **and** *zs'* :: ((*'w option*) *list*) *list*
proof (*induction zs arbitrary*: *zs'*)
case *Nil*
then show *?case*
by(*induction zs'*; *simp*)
next
fix *z* :: (*'w option*) *list*
and *zs* :: (*'w option*) *list list*
and *zs'* :: (*'w option*) *list list*
assume *ind*: $\bigwedge$*zs'*. $\forall$ *n*<*length zs*. $\forall$ *m*<*length zs'*. (*zs* ! *n* ! *m*::(*'w option*)) = *zs'* ! *m* ! *n* $\Longrightarrow$ *list-all* (λ*l*. *length l* = *length zs'*) *zs* $\Longrightarrow$ *list-all* (λ*l*. *length l* = *length zs*) *zs'* $\Longrightarrow$ *mset* (*concat zs*) = *mset* (*concat zs'*)
and *trans*: $\forall$ *n*<*length* (*z* # *zs*). $\forall$ *m*<*length zs'*. (*z* # *zs*) ! *n* ! *m* = (*zs'* ! *m* ! *n*::(*'w option*))
and *len1*: *list-all* (λ*l*. *length* (*l*::(*'w option*) *list*) = *length* (*zs'*::(*'w option*) *list list*)) (*z* # *zs*)
and *len2*: *list-all* (λ*l*. *length* (*l*::(*'w option*) *list*) = *length* ((*z*::(*'w option*) *list*) # *zs*)) *zs'*
have *trans'*: $\bigwedge$*n m*. *n*<*length* (*z* # *zs*) $\Longrightarrow$ *m*<*length zs'* $\Longrightarrow$ (*z* # *zs*) ! *n* ! *m* = (*zs'* ! *m* ! *n*::(*'w option*))
using *trans* **by** *force*
have *G1*: *mset* (*concat zs'*) = *mset* (*map hd zs'*) + *mset* (*concat* (*map tl zs'*))
using *len2*
unfolding *mset-append*[*symmetric*]
proof (*induction zs'*)
case (*Cons z zs'*) **then show** *?case* **by** (*cases z*; *simp*)
qed *simp*
have *length z* = *length zs'* **using** *len1* **by** *auto*
then have *G2*: *mset z* = *image-mset hd* (*mset zs'*)
using *trans'*[*of 0*] *len2*
proof −
have *hd-eq*: $\forall$ *l* $\in$ *set zs'*. *hd l* = *l* ! *0*
proof
fix *l* **assume** *l* $\in$ *set zs'*
then have *length l* = *Suc* (*length zs*) **using** *len2* **by** (*simp add*: *list-all-iff*)
then have *l* $\neq$ [] **by** *auto*
then show *hd l* = *l* ! *0* **by** (*simp add*: *hd-conv-nth*)
qed
have *z-eq*: *z* = *map hd zs'*
using ‹*length z* = *length zs'*› *trans'*[*of 0*] *hd-eq len2*
by (*auto intro*!: *nth-equalityI simp*: *list-all-iff*)
then show *?thesis*
by *simp*
qed
have *G3*: *mset* (*concat zs*) = *mset* (*concat* (*map tl zs'*))
proof (*intro ind*)
show $\forall$ *n*<*length zs*. $\forall$ *m*<*length* (*map tl zs'*). *zs* ! *n* ! *m* = *map tl zs'* ! *m* ! *n*
proof *safe*

```
      fix n m
      assume n < length zs m < length (map tl zs')
      then show zs ! n ! m = map tl zs' ! m ! n
        using trans'[of Suc n m] len2 by (auto simp add: nth-tl list-all-length)
    qed
  next
    show list-all (λl. length l = length (map tl zs')) zs
      using len1 by simp
  next
    show list-all (λl. length l = length zs) (map tl zs')
      using len2 by (induction zs' arbitrary: zs; simp)
  qed
  show mset (concat ((z::('w option) list) # zs)) = mset (concat zs')
    using G1 G2 G3
    by simp
qed
have fold-eq: fold (λ(a, w). (+) (Some w)) lx (Some b) = fold (λx s. s + x) (map
(Some ∘ snd) lx) (Some b) for lx :: ('a × 'w :: ref-ab-semigroup-add) list and b ::
'w :: ref-ab-semigroup-add
proof (induction lx arbitrary: b)
  case Nil
  then show ?case by simp
next
  case (Cons a lx')
  then show ?case
    by (cases a) (auto simp add: add.commute)
qed
have Hprev: ∀ a. ∃ zs zs'.
    list-split' zs (map (Some ∘ snd) (filter (λ(x, -). a = x) xs)) ∧
    list-split' zs' (map (Some ∘ snd) (filter (λ(x, -). a = x) ys)) ∧
    (∀ n m. n < length (map (Some ∘ snd) (filter (λ(x, -). a = x) xs)) ⟶ m <
length (map (Some ∘ snd) (filter (λ(x, -). a = x) ys)) ⟶ zs ! n ! m = zs' ! m !
n) ∧
    list-all (λl. length l = length (map (Some ∘ snd) (filter (λ(x, -). a = x) ys)))
zs ∧
    list-all (λl. length l = length (map (Some ∘ snd) (filter (λ(x, -). a = x) xs)))
zs'
proof
  fix a
  show ∃ zs zs'. list-split' zs (map (Some ∘ snd) (filter (λ(x, -). a = x) xs)) ∧
    list-split' zs' (map (Some ∘ snd) (filter (λ(x, -). a = x) ys)) ∧
    (∀ n m. n < length (map (Some ∘ snd) (filter (λ(x, -). a = x) xs)) ⟶ m <
length (map (Some ∘ snd) (filter (λ(x, -). a = x) ys)) ⟶ zs ! n ! m = zs' ! m !
n) ∧
    list-all (λl. length l = length (map (Some ∘ snd) (filter (λ(x, -). a = x) ys)))
zs ∧
    list-all (λl. length l = length (map (Some ∘ snd) (filter (λ(x, -). a = x) xs)))
zs'
    using wset-eq
```

proof −
show ∃ *zs zs′*. *list-split′ zs* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*)) ∧
list-split′ zs′ (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*)) ∧
(∀ *n m*. *n* < *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*)) ⟶ *m* < *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*)) ⟶ *zs* ! *n* ! *m* = *zs′* ! *m* ! *n*) ∧
list-all (λ*l*. *length l* = *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*))) *zs* ∧
list-all (λ*l*. *length l* = *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*))) *zs′*
proof (*intro list-split′-exist*[**where** *xs* = (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*)) **and** *ys* = (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*))])
show (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*) ≠ [] ∧ *map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*) ≠ []) ⟶
fold′ (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*)) = *fold′* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*))
proof
assume *H*: *map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*) ≠ [] ∧
map (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*) ≠ []
from *wset-eq* **have** *eq′*: *sum-key xs a* = *sum-key ys a*
unfolding *eq-wset-def* **by** *auto*
from *H* **obtain** *fx fxs fy fys*
where *fx-def*: *filter* (λ(*x*, -). *a* = *x*) *xs* = *fx* # *fxs*
and *fy-def*: *filter* (λ(*x*, -). *a* = *x*) *ys* = *fy* # *fys*
by (*auto simp*: *neq-Nil-conv*)
obtain *ax wx* **where** *fx-prod*: *fx* = (*ax*, *wx*) **by** (*cases fx*)
obtain *ay wy* **where** *fy-prod*: *fy* = (*ay*, *wy*) **by** (*cases fy*)
show *fold′* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*)) = *fold′* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*))
using *eq′* **unfolding** *fx-def fy-def fx-prod fy-prod*
by (*simp add*: *foldl-conv-fold fold-eq*)
qed
next
show (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *xs*) = []) = (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *ys*) = [])
proof −
have *key*: *sum-key xs a* = *sum-key ys a*
using *wset-eq* **unfolding** *eq-wset-def* **by** *simp*
have *none-iff*: (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*, -). *a* = *x*) *zs*) = []) = (*sum-key zs a* = *None*)
for *zs* :: (′*a* × ′*w* :: *ref-ab-semigroup-add*) *list* **proof** (*induction zs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons h t*)
obtain *k v* **where** *h-def*: *h* = (*k*, *v*)
by *fastforce*
then show *?case*

using *Cons fold-Some*
by (*cases a = k*) *auto*
qed
show *?thesis*
using *none-iff*[*of xs*] *none-iff*[*of ys*] *key* **by** *simp*
qed
qed
qed
qed
then have ∀ *a*. ∃ *zs zs'*. *list-split' zs* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *xs*)) ∧ *list-split' zs'* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *ys*)) ∧ *mset* (*concat zs*) = *mset* (*concat zs'*)
proof −
show ∀ *a*. ∃ *zs zs'*. *list-split' zs* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *xs*)) ∧ *list-split' zs'* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *ys*)) ∧ *mset* (*concat zs*) = *mset* (*concat zs'*)
proof (*rule allI*)
fix *a*
obtain *zs-a zs'-a* **where**
S1: *list-split' zs-a* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *xs*))
and *S2*: *list-split' zs'-a* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *ys*))
and *Strans*: ∀ *n m*. *n* < *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *xs*)) ⟶
m < *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *ys*)) ⟶
zs-a ! *n* ! *m* = *zs'-a* ! *m* ! *n*
and *Slen1*: *list-all* (λ*l*. *length l* = *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *ys*))) *zs-a*
and *Slen2*: *list-all* (λ*l*. *length l* = *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *xs*))) *zs'-a*
using *Hprev* **by** *blast*
have *Llen1*: *length zs-a* = *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *xs*))
by (*rule list-split'-length*[*OF S1*])
have *Llen2*: *length zs'-a* = *length* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *ys*))
by (*rule list-split'-length*[*OF S2*])
have *S3*: *mset* (*concat zs-a*) = *mset* (*concat zs'-a*)
using *Strans Slen1 Slen2 Llen1 Llen2 ind-help* **by** *metis*
show ∃ *zs zs'*. *list-split' zs* (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *xs*)) ∧
list-split' zs' (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *ys*)) ∧
mset (*concat zs*) = *mset* (*concat zs'*)
using *S1 S2 S3* **by** *blast*
qed
qed
then obtain *zs zs'* **where** *L1*: ⋀ *a*. *list-split'* (*zs a*) (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *xs*)) **and**
L2: ⋀ *a*. *list-split'* (*zs' a*) (*map* (*Some* ∘ *snd*) (*filter* (λ(*x*,-). *a* = *x*) *ys*)) **and** *L3*: ⋀ *a*. *mset* (*concat* (*zs a*)) = *mset* (*concat* (*zs' a*))
by *metis*

```
  have G: (∀ a. list-split' (zs a) (map (Some ∘ snd) (filter (λ(x,-). a = x) xs)))
⟹ list-split (create-split xs zs) xs
    for zs :: 'a ⇒ ('w :: ref-ab-semigroup-add) option list list and xs
    proof (induction xs arbitrary: zs)

      fix zs :: 'a ⇒ 'w option list list
      assume ∀ a. list-split' (zs (a::'a)) (map (Some ∘ snd) (filter (λ(x, y). ((λ-. a
= x)::'w ⇒ bool) y) []))
      show list-split (create-split [] zs) []
        by(auto intro: list-split.intros)
    next
      fix x :: 'a × 'w
        and xs :: ('a × 'w) list
        and zs :: 'a ⇒ 'w option list list
      assume ind1: ⋀zs. (∀ a. list-split' (zs a) (map (Some ∘ snd) (filter (λ(x, -).
a = x) xs))) ⟹ list-split (create-split xs zs) xs
       and ind2: ∀ a. list-split' (zs a) (map (Some ∘ snd) (filter (λ(x, -). a = x) (x
# xs)))
      have ind1: ⋀zs. (⋀a. list-split' (zs a) (map (Some ∘ snd) (filter (λ(x, -). a
= x) xs))) ⟹ list-split (create-split xs zs) xs
       and ind2: ⋀a. list-split' (zs a) (map (Some ∘ snd) (filter (λ(x, -). a = x)
(x # xs)))
        using ind1 ind2 by auto
      obtain x1 x2 where x-def: x = (x1,x2)
        by fastforce
      have H1: list-split (create-split xs (zs(x1 := tl (zs x1)))) xs
      proof (rule ind1)
        fix a
       show list-split' ((zs(x1 := tl (zs x1))) a) (map (Some ∘ snd) (filter (λ(x,
-). a = x) xs))
         using ind2[of a] unfolding x-def by (auto elim: list-split'.cases)
      qed
     have H2: create-split ((x1, x2) # xs) zs = map (Pair x1) (option-list (hd (zs
x1))) @ create-split xs (zs(x1 := tl (zs x1)))
        by simp
      have H3: x2 = fold' (map snd (map (Pair x1) (option-list (hd (zs x1)))))
      proof (cases zs x1)
        case Nil
        then show ?thesis
          using ind2[of x1] unfolding x-def using list-split'.cases by fastforce
      next
        case (Cons h t)
        then show ?thesis
          using ind2[of x1] unfolding x-def comp-def
        proof −
          assume hcons: zs x1 = h # t
            and ls: list-split' (zs x1) (map (λx. Some (snd x)) (filter (λ(x, -). x1 =
x) ((x1, x2) # xs)))
           have filt: map (λx. Some (snd x)) (filter (λ(x, -). x1 = x) ((x1, x2) #
```

```
xs)) = Some x2 # map (λx. Some (snd x)) (filter (λ(x, -). x1 = x) xs)
          by simp
        have list-split' (h # t) (Some x2 # map (λx. Some (snd x)) (filter (λ(x,
-). x1 = x) xs))
          using ls hcons filt by simp
        then have fold-h: Some x2 = fold' h and h-ne: h ≠ []
          by (auto elim: list-split'.cases)
        have x2 = fold' (option-list h)
          by (rule fold-option[OF fold-h h-ne])
       then show x2 = fold' (map snd (map (Pair x1) (option-list (hd (zs x1)))))
          by (simp add: hcons comp-def)
      qed
    qed
    have H4: map (Pair x1) (option-list (hd (zs x1))) ≠ []
    proof (cases zs x1)
      case Nil
      then show ?thesis
        using ind2[of x1] unfolding x-def using list-split'.cases by fastforce
    next
      case (Cons h t)
      then show ?thesis
        using ind2[of x1] unfolding x-def
        by (auto elim: list-split'.cases simp add: fold-option-not-none)
    qed
   have list-all (λ(a, b). a = x1) (map (Pair x1) (option-list l)) for l :: 'c option
list
      proof (induction l)
        case (Cons a l) then show ?case by (cases a) auto
      qed simp
    then have H5: list-all (λ(a, b). a = x1) (map (Pair x1) (option-list (hd (zs
x1))))
      by auto
    show list-split (create-split (x # xs) zs) (x # xs)
      using H1 H2 H3 H4 H5 Split
      unfolding x-def by fast
  qed
 have G3: mset (create-split xs zs) = mset (create-split ys zs')
 proof (rule multiset-eqI)
  fix x
  show count (mset (create-split xs zs)) x = count (mset (create-split ys zs')) x
    using L1[THEN list-split'-length] L2[THEN list-split'-length]
    by (cases x) (auto simp: L3 simp flip: create-split-count)
 qed
 show ?thesis
  using G L1 L2 G3
  by blast
qed

lemma w-size-eq-Suc-imp-eq-union:
```

```
  assumes H: size M = Suc n
  shows ∃x w N. M = wupdate N x (Some w) ∧ weight N x = None
proof −
  have H1: ∃x w. weight M x = Some w
  proof −
    obtain l where M-eq: M = abs-wset l using get-abs-wset by blast
    obtain m1 m2 l' where l-eq: l = (m1, m2) # l'
    proof (cases l)
      case Nil
      then have wset M = {} by (simp add: M-eq wset.abs-eq)
      then have size M = 0 by (simp add: size-wset-overloaded-def)
      with H show ?thesis by simp
    next
      case (Cons m l')
      obtain m1 m2 where m = (m1, m2) by (cases m)
      then show ?thesis using that Cons by blast
    qed
     have not-none: fold (λ(a, w). (+) (Some w)) (filter (λ(e', -). m1 = e') l')
(Some m2) ≠ None
      by (rule fold-Some)
     obtain v where fold (λ(a, w). (+) (Some w)) (filter (λ(e', -). m1 = e') l')
(Some m2) = Some v
      using not-none by (cases fold (λ(a, w). (+) (Some w)) (filter (λ(e', -). m1
= e') l') (Some m2)) auto
    then show ?thesis
      by (auto simp add: M-eq l-eq weight.abs-eq)
  qed
  then obtain x w where Hxw: weight M x = Some w by blast
  show ?thesis
  proof (intro exI conjI)
    show M = wupdate (wupdate M x None) x (Some w)
      by (simp add: weight-eq-iff[symmetric] Hxw)
    show weight (wupdate M x None) x = None
      by simp
  qed
qed

lemma size-wupdate:
  assumes size: Suc k = size (wupdate N x (Some w))
    and weight: weight N x = None
  shows k = size N
proof −
  obtain l where N-def: N = abs-wset l
    by (metis get-abs-wset)
  have list-all (λ(a, -). x ≠ a) l
    using weight
    unfolding N-def weight.abs-eq
  proof (induction l)
    case Nil then show ?case by simp
```

```
  next
    case (Cons h t) then show ?case
      by (metis (mono-tags, lifting) filter.simps(2) fold-Some(1) fold-simps(2)
          list.pred-inject(2) plus-option-simps(1) split-beta)
  qed
  then have set-eq: {y. (∃ b. (y, b) ∈ set l) ∧ x ≠ y} − {x} = {y. ∃ x∈set l. y =
fst x}
    by(auto simp add: Bex-def Ball-set-list-all[symmetric])
  have fin: finite {y. (∃ b. (y, b) ∈ set l) ∧ x ≠ y}
  proof −
    have finite (Domain (set l))
      using finite-Domain by blast
    then show ?thesis
      by (simp add: Domain-unfold)
  qed
  show ?thesis
    using size set-eq fin
    unfolding N-def size-wset-overloaded-def
   by(auto simp add: wset.abs-eq wupdate.abs-eq weight.abs-eq image-def card.insert-remove)
qed

theorem wset-induct [case-names empty add, induct type: multiset]:
  assumes empty: ⋀ M. M = wempty ⟹ P M
    and add: ⋀x w M. P M ⟹ weight M x = None ⟹ P (wupdate M x (Some
w))
  shows P M
proof (induct size M arbitrary: M)
  fix M :: ('a, 'b) wset
  assume size: 0 = size M
  obtain l where M-def: M = abs-wset l
    using get-abs-wset
    by auto
  show P M
  proof (rule empty)
    show M = wempty
      using size
      unfolding weight-eq-iff[symmetric]
      by (auto simp add: M-def size-wset-overloaded-def wset.abs-eq weight.abs-eq)
  qed
next
  fix k :: nat
    and M :: ('a, 'b) wset
  assume ind: ⋀M. k = size M ⟹ P M
    and size: Suc k = size M
  obtain N x w where M-def: M = wupdate N x (Some w) and weight-N: weight
N x = None
    using size[symmetric] w-size-eq-Suc-imp-eq-union
    by blast
  show P M
```

using *size weight-N ind*[*of N*] *size-wupdate add*
unfolding *M-def*
by *metis*
qed

5 The Negative Representation

lemma *weight-inj*: *inj weight*
unfolding *inj-def* **by** *transfer* (*auto simp*: *eq-wset-def fun-eq-iff*)

lemma *type-definition-wset*: *type-definition weight* (*inv weight*) {*f* :: *'a* ⇒ (*'w* :: *ref-ab-semigroup-add*) *option*. *finite* {*x*. *f x* ≠ *None*}}
proof *standard*
fix *x* :: (*'a*, *'w*) *wset*
have *H1* : *finite* {*e*. *list-ex* (λ*x*. *fst x* = *e*) *kxs*} **for** *kxs* :: (*'a* × *'w* :: *ref-ab-semigroup-add*) *list*
by(*induction kxs*; *simp*)
have *H2*: {*e*. ∃ *y*. *sum-key kxs e* = *Some y*} ⊆ {*e*. *list-ex* (λ*x*. *fst x* = *e*) *kxs*}
for *kxs* :: (*'a* × *'w* :: *ref-ab-semigroup-add*) *list*
by (*induction kxs*) *auto*

show *weight x* ∈ {*f*. *finite* {*x*. *f x* ≠ *None*}}
using *H1*[*of* (*rep-wset x*)] *H2*[*of* (*rep-wset x*)]
by (*auto simp add*: *weight.rep-eq intro*: *rev-finite-subset*)
next
fix *x* :: (*'a*, *'w*) *wset*
show *inv weight* (*weight x*) = *x*
by (*rule inv-f-f*, *rule weight-inj*)
next
fix *f* :: *'a* ⇒ *'w option*
assume *fin*: *f* ∈ {*f*. *finite* {*x*. *f x* ≠ *None*}}
show *weight* (*inv weight f*) = *f*
proof (*rule f-inv-into-f*)
show *f* ∈ *range weight*
proof −
have ∃ *x*. *f* = *weight x*
proof (*rule finite-Map-induct*[*of f* λ*f*. ∃ *x*. *f* = *weight x*])
show *finite* (*dom f*)
using *fin* **unfolding** *dom-def* **by** *simp*
next
show ∃ *x*. (λ-. *None*) = *weight x*
by (*rule exI*[**where** *x* = *wempty*]) *simp*
next
fix *x* :: *'a* **and** *w* :: *'w* **and** *f'* :: *'a* ⇒ *'w option*
assume *finite* (*dom f'*) *x* ∉ *dom f'* **and** *IH*: ∃ *ws*. *f'* = *weight ws*
then obtain *ws* **where** *ws-def*: *f'* = *weight ws* **by** *blast*
show ∃ *x2*. *f'*(*x* ↦ *w*) = *weight x2*
proof (*rule exI*[**where** *x* = *wupdate ws x* (*Some w*)])
show *f'*(*x* ↦ *w*) = *weight* (*wupdate ws x* (*Some w*))

```
          by (simp add: ws-def)
        qed
      qed
      then show ?thesis by (simp add: image-def)
    qed
  qed
qed

lemma weight-finite: finite {x. ∃ y. weight M x = Some y}
  using type-definition-wset
  unfolding type-definition-def by auto

lemma wadd-assoc: wadd x (wadd y z) = wadd (wadd x y) z
  by transfer auto

lemma wadd-commute: wadd x y = wadd y x
  by transfer (auto simp: eq-wset-append-sym)

lemma wadd-wsingle[simp]: wadd (wsingle x w) ws = wupdate ws x (Some w +
weight ws x)
  unfolding weight-eq-iff[symmetric] by auto

lemma w-list-all2-split-left-invariance:
  list-all2 (rel-prod R (=)) xs ys ⟹ list-split xs' xs ⟹
  ∃ ys'. list-all2 (rel-prod R (=)) xs' ys' ∧ list-split ys' ys
proof (induction xs arbitrary: ys xs')
  fix ys :: ('c × 'b) list
    and xs' :: ('a × 'b) list
  assume list-all2 (rel-prod R (=)) [] ys
    and list-split xs' []
  then show ∃ ys'. list-all2 (rel-prod R (=)) xs' ys' ∧ list-split ys' ys
    by(cases xs'; auto elim: list-split.cases)
next
  fix x :: 'a × 'b
    and xs :: ('a × 'b) list
    and ys :: ('c × 'b) list
    and xs' :: ('a × 'b) list
  assume ind: ⋀ys (xs' :: ('a × 'b) list). list-all2 (rel-prod R (=)) xs ys ⟹ list-split
xs' xs ⟹ ∃ ys'. list-all2 (rel-prod R (=)) xs' ys' ∧ list-split ys' ys
    and list2-all': list-all2 (rel-prod R (=)) (x # xs) ys
    and list-split': list-split xs' (x # xs)
  have ys ≠ []
    using list2-all'
    by auto
  then obtain x' wx y' wy yss where ys-def: ys = (y', wy) # yss and x-def: x
= (x', wx)
    using list2-all'
    by (metis list.exhaust surj-pair)
  have wx-def: wx = wy
```

using *list2-all′ x-def ys-def*
by *fastforce*
have *R-x-y*: *R x′ y′*
using *list2-all′ x-def ys-def*
by *simp*
obtain *exs exs′* **where** *list-split′*: *list-split exs′ xs* **and** *xs′-def*: *xs′ = exs @ exs′* **and**
wy-fold: *wy = fold′ (map snd exs)* **and** *exs-nonempty*: *exs ≠ []* **and** *list-all-exs*: *list-all (λ(a,b). a = x′) exs*
using *list-split′ x-def wx-def*
by(*auto elim*: *list-split.cases*)
obtain *eys′* **where** *ind-e*: *list-all2 (rel-prod R (=)) exs′ eys′* **and** *wset-yss*: *list-split eys′ yss*
using *list2-all′ list-split′ x-def ys-def ind eq-wset-sym*
by *fastforce*
have *exs ≠ [] ⟹ list-split eys′ yss ⟹ list-split (map (λ(-, w). (y′, w)) exs @ eys′) ((y′, fold′ (map snd exs)) # yss)*
proof (*induction length exs arbitrary*: *exs*)
case *0*
then show *?case*
by *simp*
next
fix *n* :: *nat*
and *exs* :: (*′a* × *′b*) *list*
assume *ind*: ⋀*exs*. *n = length* (*exs*::(*′a* × *′b*) *list*) ⟹ *exs ≠ []* ⟹ *list-split eys′ yss* ⟹ *list-split* (*map* (λ(-, *y*). (*y′*, *y*)) *exs* @ *eys′*) ((*y′*, *fold′* (*map snd exs*)) # *yss*)
and *length*: *Suc n = length* (*exs*::(*′a* × *′b*) *list*)
and *non-empty*: (*exs*::(*′a* × *′b*) *list*) ≠ []
and *eq*: *list-split eys′ yss*
obtain *e11 e12 exs′* **where** *exs-def* : *exs* = (*e11*,*e12*) # *exs′*
by (*metis list.exhaust old.prod.exhaust non-empty*)
consider *exs′* = [] | *exs′ ≠* []
by *fast*
then show *list-split* (*map* (λ(-, *y*). (*y′*, *y*)) (*exs*::(*′a* × *′b*) *list*) @ *eys′*) ((*y′*, *fold′* (*map snd exs*)) # *yss*)
proof (*cases*, *goal-cases nil not-nil′*)
case *nil*
then show *?case*
unfolding *exs-def*
by(*auto simp add*: *foldl-assoc add.assoc eq-wset-sym eq intro*!: *eq-wset-elem-remove list-split-cons-eq*)
next
case *not-nil′*
then obtain *e21 e22 exs″* **where** *exs′-def* : *exs′* = (*e21*,*e22*) # *exs″*
by (*metis list.exhaust old.prod.exhaust*)
have *H-len*: *n = length* ((*e11*, *e12*+*e22*) # *exs″*)
using *length* **unfolding** *exs-def exs′-def* **by** *simp*
have *H-ih*: *list-split* (*map* (λ(-, *y*). (*y′*, *y*)) ((*e11*, *e12*+*e22*) # *exs″*) @ *eys′*)

((y′, fold′ (map snd ((e11, e12+e22) # exs′′))) # yss)
using ind[of (e11, e12+e22) # exs′′] H-len eq by simp
have H-comb: list-split ((y′, e12) # (y′, e22) # map (λ(-, y). (y′, y)) exs′′ @ eys′) ((y′, e12+e22) # map (λ(-, y). (y′, y)) exs′′ @ eys′)
using list-split-comb[of y′ e12 e22 map (λ(-, y). (y′, y)) exs′′ @ eys′] by simp
show ?case
unfolding exs-def exs′-def
using list-split-trans[OF H-comb] H-ih
by (simp add: foldl-assoc[of (+)] add.assoc)
qed
qed
then have list-split (map (λ(-, w). (y′, w)) exs @ eys′) ((y′, wy) # yss)
using exs-nonempty wset-yss
by(simp add: wx-def wy-fold wset.abs-eq-iff)
also have list-all2 (rel-prod R (=)) xs′ (map (λ(-, w). (y′, w)) exs @ eys′)
using R-x-y list-all-exs ind-e
unfolding xs′-def
by (induction exs) auto
ultimately show ∃ys′. list-all2 (rel-prod R (=)) xs′ ys′ ∧ list-split ys′ ys
unfolding ys-def x-def
by (intro exI[where x = (map (λ(-,w). (y′,w)) exs) @ eys′]) auto
qed

lemma w-list-all2-split-right-invariance:
list-all2 (rel-prod R (=)) xs ys ⟹ list-split ys′ ys ⟹
∃xs′. list-all2 (rel-prod R (=)) xs′ ys′ ∧ list-split xs′ xs
using w-list-all2-split-left-invariance list.rel-flip
by (metis conversep-eq prod.rel-conversep)

lemma w-list-all2-reorder-left-invariance:
list-all2 (rel-prod R (=)) xs ys ⟹ list-split xs′ xs ⟹
∃ys′. list-all2 (rel-prod R (=)) xs′ ys′ ∧ eq-wset ys′ ys
using w-list-all2-split-left-invariance list-split-eq-wset
by metis

lemma w-list-all2-reorder-right-invariance:
list-all2 (rel-prod R (=)) xs ys ⟹ list-split ys′ ys ⟹
∃xs′. list-all2 (rel-prod R (=)) xs′ ys′ ∧ eq-wset xs′ xs
using w-list-all2-reorder-left-invariance list.rel-flip
by (metis conversep-eq prod.rel-conversep)

lemma eq-wset-remove1: ListMem x xs ⟹ eq-wset (x # (remove1 x xs)) xs
proof(induction length xs arbitrary: x xs)
case 0
then show ?case
by(auto elim: ListMem.cases)
next
case (Suc n)

```
  then show ?case
  proof (cases xs)
    case Nil
    with Suc show ?thesis by auto
  next
    case (Cons x' xs')
    then show ?thesis
    proof (cases x = x')
      case True
      then show ?thesis
        unfolding eq-wset-def
        using Cons by simp
    next
      case False
      then show ?thesis
        using Cons Suc.hyps(1) eq-wset-elem-switch eq-wset-elem-remove
        using ListMem-iff[of x xs] Suc.prems eq-wset-elem-back[of - x]
         eq-wset-elem-back'[of x remove1 x xs] eq-wset-sym[of xs remove1 x xs @ [x]]
          eq-wset-trans[of x # remove1 x xs remove1 x xs @ [x] xs]
          remove1-split[of x xs remove1 x xs] by fastforce
    qed
  qed
qed

lemma wset-mset-list:
  mset (xs :: ('a × 'w :: ref-ab-semigroup-add) list) = mset ys ⟹
  abs-wset xs = abs-wset ys
proof (induction xs arbitrary: ys)
  fix ys :: ('a × 'w) list
  assume mset [] = mset ys
  then show abs-wset [] = abs-wset ys
    by force
next
  fix x :: 'a × 'w
    and xs :: ('a × 'w) list
    and ys :: ('a × 'w) list
  obtain x' w where x-def: x = (x', w)
    by force
  assume ind: ⋀ys. mset xs = mset ys ⟹ abs-wset xs = abs-wset ys
    and mset-eq: mset (x # xs) = mset ys
  have H: ListMem (x', w) ys
    using mset-eq
    by (metis ListMem-iff list.set-intros(1) set-mset-mset x-def)
  then have wset-ys: abs-wset ys = wadd (wsingle x' w) (abs-wset (remove1 (x',
w) ys))
   by(simp add: wsingle.abs-eq wadd.abs-eq wset.abs-eq-iff eq-wset-remove1 eq-wset-sym)
  have wset-xs: eq-wset xs (remove1 (x', w) ys)
    using ind mset-eq
    by (metis mset-remove1 remove1.simps(2) x-def wset.abs-eq-iff[symmetric])
```

show *abs-wset* (*x* # *xs*) = *abs-wset ys*
by(*auto simp add*: *wset-ys wset-xs x-def wsingle.abs-eq wadd.abs-eq wset.abs-eq-iff intro*!: *eq-wset-elem-remove*)
qed

lemma *fold-some'*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*x* # *xs*) *w* ≠ *None*
by (*induction xs arbitrary*: *x w*) (*auto simp*: *plus-option-case*)

lemma *eq-wset-mset*:
mset (*xs* :: ('*a* × '*w* :: *ref-ab-semigroup-add*) *list*) = *mset ys* ⟹ *eq-wset xs ys*
proof (*induction xs arbitrary*: *ys*)
fix *ys* :: ('*a* × '*w*) *list*
assume *mset* [] = *mset ys*
then show *eq-wset* [] *ys*
unfolding *eq-wset-def*
by *force*
next
fix *x* :: '*a* × '*w*
and *xs* :: ('*a* × '*w*) *list*
and *ys* :: ('*a* × '*w*) *list*
obtain *x'* *w* **where** *x-def*: *x* = (*x'*, *w*)
by *force*
assume *ind*: ⋀*ys*. *mset xs* = *mset ys* ⟹ *eq-wset xs ys*
and *mset-eq*: *mset* (*x* # *xs*) = *mset ys*
have *ListMem* (*x'*, *w*) *ys*
using *mset-eq*
by (*metis ListMem-iff list.set-intros*(*1*) *set-mset-mset x-def*)
then have *wset-ys*: *eq-wset ys* ((*x'*, *w*) # (*remove1* (*x'*, *w*) *ys*))
using *eq-wset-remove1 eq-wset-sym*
by *blast*
have *wset-xs*: *eq-wset xs* (*remove1* (*x'*, *w*) *ys*)
using *ind mset-eq*
by (*metis mset-remove1 remove1.simps*(*2*) *x-def*)
show *eq-wset* (*x* # *xs*) *ys*
unfolding *x-def*
using *wset-ys wset-xs eq-wset-sym eq-wset-elem-remove*
using *eq-wset-trans* **by** *blast*
qed

lemma *eq-wset-set-fst*:
assumes *A*: *eq-wset xs ys*
shows *fst* ‘ *set xs* = *fst* ‘ *set ys*
proof −
have *H1*: (∃ *x* ∈ *set xs*. *a* = *fst x*) = (*fold* (λ(-, *w*). (+) (*Some w*)) (*filter* (λ(*a'*,-). *a* = *a'*) *xs*) *None* ≠ *None*) **for** *a* :: '*a* **and** *xs* :: ('*a* × '*w* :: *ref-ab-semigroup-add*) *list*
proof (*cases filter* (λ(*a'*, -). *a* = *a'*) *xs*)
case *Nil*
then have *no-match*: ∀ *x*∈*set xs*. *a* ≠ *fst x*

```
    by (simp add: filter-empty-conv split-beta)
  then have sum-key xs a = None
    by (simp add: Nil)
  then show ?thesis using no-match by simp
 next
  case (Cons x xs')
  then show ?thesis
  proof (cases x)
   case (Pair a' w)
   then show ?thesis
    using Cons by (auto simp: fold-Some' filter-eq-Cons-iff)
  qed
 qed
 show ?thesis
  using A
  unfolding image-def eq-wset-def
  by(auto simp: eq-wset-def image-def H1 fold-Some')
qed
```

6 BNF Registration

```
lift-bnf ('a, dead 'w :: ref-ab-semigroup-add) wset
 for map: wimage rel: wrel
proof −
 fix Ps :: 'a ⇒ 'b ⇒ bool and Qs :: 'b ⇒ 'c ⇒ bool
 assume Ps OO Qs ≠ bot
 show list-all2 (rel-prod Ps (=)) OO eq-wset OO list-all2 (rel-prod Qs (=)) ≤
      (eq-wset :: ('a × 'w) list ⇒ -) OO list-all2 (rel-prod (Ps OO Qs) (=)) OO
(eq-wset :: ('c × 'w) list ⇒ -)
 proof safe
  fix xs :: ('a × 'w :: ref-ab-semigroup-add) list and zs :: ('c × 'w) list
   and ys :: ('b × 'w) list and y's :: ('b × 'w) list
  assume lall-Ps: list-all2 (rel-prod Ps (=)) xs ys
   and eq-ys: eq-wset ys y's
   and lall-Qs: list-all2 (rel-prod Qs (=)) y's zs
  obtain ys' y's' where
   split-ys: list-split ys' ys and split-y's: list-split y's' y's
   and mset-eq: mset ys' = mset y's'
   using list-split-exist[OF eq-ys] by blast
  obtain xs' where xs'-lall: list-all2 (rel-prod Ps (=)) xs' ys'
   and xs'-eq: eq-wset xs' xs
   using w-list-all2-reorder-right-invariance[OF lall-Ps split-ys] by blast
  obtain zs' where zs'-lall: list-all2 (rel-prod Qs (=)) y's' zs'
   and zs'-eq: eq-wset zs' zs
   using w-list-all2-reorder-left-invariance[OF lall-Qs split-y's] by blast
  obtain ys'' where ys''-lall: list-all2 (rel-prod Qs (=)) ys' ys''
   and ys''-mset: mset ys'' = mset zs'
   using list-all2-reorder-left-invariance[OF zs'-lall mset-eq] by blast
  show (eq-wset OO list-all2 (rel-prod (Ps OO Qs) (=)) OO eq-wset) xs zs
```

proof (*intro relcomppI*)
show *eq-wset xs xs'*
using *xs'-eq eq-wset-sym* **by** *blast*
show *eq-wset ys'' zs*
using *eq-wset-mset eq-wset-trans ys''-mset zs'-eq* **by** *blast*
show *list-all2* (*rel-prod* (*Ps OO Qs*) (=)) *xs' ys''*
by (*smt* (*verit*, *best*) *list-all2-trans rel-prod-sel relcomppI xs'-lall ys''-lall*)
qed
qed
next
have *H*: ⋃ (*Basic-BNFs.fsts* ‘ *set xs*) = (*set o* (*map fst*)) *xs* **for** *xs* :: (*'a* × *'w* :: *ref-ab-semigroup-add*) *list*
by (*induction xs*) *auto*
show (*Ss* :: *'a set set*) ≠ {} ⟹ ⋂ *Ss* ≠ {} ⟹
(⋂ *As*∈*Ss*. {(*x*, *x'*). *eq-wset x x'*} `` {*x* :: (*'a* × *'w*) *list*. ⋃ (*Basic-BNFs.fsts* ‘ *set x*) ⊆ *As*}) ⊆ {(*x*, *x'*). *eq-wset x x'*} `` {*x*. ⋃ (*Basic-BNFs.fsts* ‘ *set x*) ⊆ ⋂ *Ss*} **for** *Ss*
unfolding *H*
using *Inter-greatest*[*of Ss* (*set o map fst*) -]
unfolding *subset-eq Ball-def Bex-def INT-iff Image-iff mem-Collect-eq prod.case*
by (*metis comp-apply eq-wset-fst eq-wset-refl*)
qed

7 Further Operations

lift-definition *wfilter* :: ‹(*'a* ⇒ *bool*) ⇒ (*'a*, *'w* :: *ref-ab-semigroup-add*) *wset* ⇒ (*'a*, *'w* :: *ref-ab-semigroup-add*) *wset*› **is**
‹λ*f l*. *filter* (λ*x*. *f* (*fst x*)) *l*›
unfolding *eq-wset-def*
proof (*safe*)
fix *f* :: *'a* ⇒ *bool* **and** *l1 l2* :: (*'a* × *'w*) *list* **and** *q* :: *'a*
assume *H*: ∀ *q*. *sum-key l1 q* = *sum-key l2 q*
show *sum-key* (*filter* (λ*x*. *f* (*fst x*)) *l1*) *q* = *sum-key* (*filter* (λ*x*. *f* (*fst x*)) *l2*) *q*
proof (*cases f q*)
case *True*
then show *?thesis*
using *H*[*rule-format*, *of q*]
unfolding *filter-filter*
by (*metis* (*mono-tags*, *lifting*) *case-prod-beta filter-cong*)
next
case *False*
then show *?thesis*
unfolding *filter-filter*
by (*metis* (*mono-tags*, *lifting*) *case-prod-beta filter-False*)
qed
qed

definition *wimage-option* :: ‹(*'a* ⇒ *'b option*) ⇒ (*'a*, *'w* :: *ref-ab-semigroup-add*) *wset* ⇒ (*'b*, *'w* :: *ref-ab-semigroup-add*) *wset*› **where**

wimage-option f ws = wimage ((case-option (SOME x. True) id) o f) (wfilter ((case-option False (λ-. True)) o f) ws)

lemma *rep-wset-wempty*[*simp*]: *rep-wset wempty = ([] :: ('a × ('b :: ref-ab-semigroup-add)) list)*
unfolding *wempty-def*
using *Quotient-rep-abs*[*OF Quotient-wset, of* [] :: ('*a* × '*b*) *list*]
by (*cases rep-wset* (*abs-wset* ([] :: ('*a* × '*b*) *list*))) *auto*

lemma *wadd-wsingle-wempty*[*simp*]: *wadd* (*wsingle x w*) *wempty* = *wsingle x w*
by *transfer simp*

lemma *wempty-if-None*: (⋀*x*. *weight w x* = *None*) ⟹ *w* = *wempty*
by *transfer* (*simp add*: *eq-wset-def*)

8 Switching Between Representations

locale *wset-as-pfun* **begin**
setup-lifting *type-definition-wset*

lemma *wempty-transfer*[*transfer-rule*]: *pcr-wset R1 R2* (λ*x*. *None*) *wempty*
unfolding *wempty-def pcr-wset-def cr-wset-def option.rel-eq fun.rel-eq eq-OO weight.abs-eq*
unfolding *rel-fun-def*
by(*rule relcomppI*[*of - - weight* (*abs-wset* []); (*simp add*: *weight.abs-eq*)*?*)

lemma *weight-transfer*[*transfer-rule*]: *rel-fun* (*pcr-wset* (=) (=)) (*rel-fun* (=) (=)) (λ*ws x*. *ws x*) *weight*
unfolding *wsingle-def pcr-wset-def cr-wset-def option.rel-eq fun.rel-eq eq-OO map-fun-def comp-def*
unfolding *rel-fun-def*
by *simp*

lemma *in-set-conv-sum-key-Some*: *x* ∈ *fst* ' *set kvs* ⟷ (∃ *v*. *sum-key kvs x* = *Some v*)
proof (*induction kvs*)
case *Nil*
show *?case* **by** *simp*
next
case (*Cons kv kvs*)
obtain *k v* **where** *kv-def*: *kv* = (*k*, *v*) **by** (*cases kv*)
show *?case*
proof (*cases k* = *x*)
case *True*
then show *?thesis*
by (*auto simp*: *kv-def fold-Some'*)
next
case *False*
then show *?thesis*

using *Cons.IH* **by** (*auto simp*: *kv-def*)
qed
qed

lemma *wset-transfer*[*transfer-rule*]: *rel-fun* (*pcr-wset* (=) (=)) (=) (λ*ws*. {*x*. *ws x* ≠ *None*}) *wset*
by (*auto simp*: *rel-fun-def pcr-wset-def cr-wset-def option.rel-eq wset.rep-eq weight.rep-eq in-set-conv-sum-key-Some del*: *imageE*)

lemma *wsingle-transfer*[*transfer-rule*]: *rel-fun* (=) (*rel-fun* (=) (*pcr-wset* (=) (=))) (λ*x w x'*. *if x* = *x' then Some w else None*) *wsingle*
by (*auto simp*: *rel-fun-def pcr-wset-def cr-wset-def option.rel-eq relcompp-apply*)

lemma *wadd-transfer*[*transfer-rule*]: *rel-fun* (*pcr-wset* (=) (=)) (*rel-fun* (*pcr-wset* (=) (=)) (*pcr-wset* (=) (=))) (λ*ws ws' x*. *ws x* + *ws' x*) *wadd*
by (*auto simp*: *rel-fun-def pcr-wset-def cr-wset-def option.rel-eq relcompp-apply*)

lemma *wupdate-transfer*[*transfer-rule*]: *rel-fun* (*pcr-wset* (=) (=)) (*rel-fun* (=) (*rel-fun* (=) (*pcr-wset* (=) (=)))) (λ*ws x w x'*. *if x* = *x' then w else ws x'*) *wupdate*
by (*auto simp*: *rel-fun-def pcr-wset-def cr-wset-def option.rel-eq relcompp-apply*)

lemma *fold-eq-wset*: *eq-wset l l'* ⟹ *sum-key* (*map* (*map-prod f id*) *l'*) *x* = *sum-key* (*map* (*map-prod f id*) *l*) *x*
proof −
assume *A*: *eq-wset l l'*
have *H2*: *sum-key* (*map* (*map-prod f id*) *l*) *x* =
sum-key (*map* (*map-prod f id*) (*filter g l*)) *x* +
sum-key (*map* (*map-prod f id*) (*filter* (λ*x*. ¬ *g x*) *l*)) *x* **for** *f* :: *'a* ⇒ *'c* **and** *g* :: *'a* × *'b* ⇒ *bool* **and** *l* :: (*'a* × *'b* :: *ref-ab-semigroup-add*) *list* **and** *x* :: *'c*
proof (*induction l*)
show *sum-key* (*map* (*map-prod f id*) []) *x* = *sum-key* (*map* (*map-prod f id*) (*filter g* [])) *x* + *sum-key* (*map* (*map-prod f id*) (*filter* (λ*x*. ¬ *g x*) [])) *x*
by *simp*
next
fix *a* :: *'a* × *'b*
and *l* :: (*'a* × *'b* :: *ref-ab-semigroup-add*) *list*
assume *ind*: *fold* (λ*a*. *case a of* (*a*, *w*) ⇒ (+) (*Some w*)) (*filter* (λ*a*. *case a of* (*a'*, *uu*-) ⇒ *x* = *a'*) (*map* (*map-prod f id*) *l*)) *None* = *fold* (λ*a*. *case a of* (*a*, *w*) ⇒ (+) (*Some w*)) (*filter* (λ*a*. *case a of* (*a'*, *uu*-) ⇒ *x* = *a'*) (*map* (*map-prod f id*) (*filter g l*))) *None* + *fold* (λ*a*. *case a of* (*a*, *w*) ⇒ (+) (*Some w*)) (*filter* (λ*a*. *case a of* (*a'*, *uu*-) ⇒ *x* = *a'*) (*map* (*map-prod f id*) (*filter* (λ*x*. ¬ *g x*) *l*))) *None*
then show *fold* (λ*a*. *case a of* (*a*, *w*) ⇒ (+) (*Some w*)) (*filter* (λ*a*. *case a of* (*a'*, *uu*-) ⇒ *x* = *a'*) (*map* (*map-prod f id*) (*a* # *l*))) *None* = *fold* (λ*a*. *case a of* (*a*, *w*) ⇒ (+) (*Some w*)) (*filter* (λ*a*. *case a of* (*a'*, *uu*-) ⇒ *x* = *a'*) (*map* (*map-prod f id*) (*filter g* (*a* # *l*)))) *None* + *fold* (λ*a*. *case a of* (*a*, *w*) ⇒ (+) (*Some w*)) (*filter* (λ*a*. *case a of* (*a'*, *uu*-) ⇒ *x* = *a'*) (*map* (*map-prod f id*) (*filter* (λ*x*. ¬ *g x*) (*a* # *l*)))) *None*
by(*auto simp add*: *fold-Some-out add.commute*[*symmetric*] *add.assoc*[*symmetric*])
qed

have *H3'*: $(\lambda p.\ (case\ p\ of\ (a',\ uu\text{-}) \Rightarrow a = a') \wedge (case\ map\text{-}prod\ f\ id\ p\ of\ (a',\ uu\text{-}) \Rightarrow x = a')) = (\lambda(a',\ \text{-}).\ a = a' \wedge f\ a = x)$ **for** $x :: 'c$ **and** $f :: 'a \Rightarrow 'c$ **and** $a :: 'a$
by *force*
have *H3''*: $(\lambda x.\ case\ map\text{-}prod\ f\ id\ x\ of\ (a,\ w) \Rightarrow (+)\ (Some\ w)) = (\lambda(a,\ w) \Rightarrow (+)\ (Some\ w))$ **for** $f :: 'a \Rightarrow 'c$
by *fastforce*
have *H3*: *fold* $(\lambda(a,\ w).\ (+)\ (Some\ w))$ (*filter* $(\lambda(a',\ \text{-}).\ x = a')$ (*map* (*map-prod f id*) (*filter* $(\lambda(a',\ \text{-}).\ a = a')\ l$))) *None* =
fold $(\lambda(a,\ w).\ (+)\ (Some\ w))$ (*filter* $(\lambda(a',\ \text{-}).\ a = a' \wedge f\ a = x)\ l$) *None*
for $x :: 'c$ **and** $f :: 'a \Rightarrow 'c$ **and** $a :: 'a$ **and** $l :: ('a \times 'b :: ref\text{-}ab\text{-}semigroup\text{-}add)\ list$
unfolding *filter-map comp-def filter-filter fold-map H3' H3''*
by *simp*
have *eq-wset l l'* $\Longrightarrow$ *sum-key* (*map* (*map-prod f id*) *l'*) *x* = *sum-key* (*map* (*map-prod f id*) *l*) *x*
proof (*induction map fst l arbitrary*: *l l' rule*: *length-induct*)
fix $l :: ('a \times 'b)\ list$
and $l' :: ('a \times 'b)\ list$
assume *ind*: $\forall ys.\ length\ ys < length\ (map\ fst\ l) \longrightarrow (\forall (la :: ('a \times 'b)\ list).\ ys = map\ fst\ la \longrightarrow (\forall lb.\ eq\text{-}wset\ la\ lb \longrightarrow$
sum-key (*map* (*map-prod f id*) *lb*) *x* = *sum-key* (*map* (*map-prod f id*) *la*) *x*))
and *l-l'-eq*: *eq-wset* $(l::('a \times 'b)\ list)\ l'$
have *ind*: $\bigwedge(l' :: ('a \times 'b)\ list)\ l''.\ length\ (map\ fst\ l') < length\ (map\ fst\ l) \Longrightarrow eq\text{-}wset\ l'\ l'' \Longrightarrow$
sum-key (*map* (*map-prod f id*) *l''*) *x* = *sum-key* (*map* (*map-prod f id*) *l'*) *x*
using *ind*
by *metis*
consider $l = [] \mid l \neq []$
by *auto*
then show *fold* $(\lambda(a,\ w).\ (+)\ (Some\ (w::'b)))$ (*filter* $(\lambda(a',\ \text{-}).\ x = a')$ (*map* (*map-prod f id*) *l'*)) *None* = *fold* $(\lambda(a,\ w).\ (+)\ (Some\ w))$ (*filter* $(\lambda(a',\ \text{-}).\ x = a')$ (*map* (*map-prod f id*) *l*)) *None*
proof (*cases*, *goal-cases nil app*)
case *nil*
have *l'-def*: $l' = []$
using *l-l'-eq fold-Some*(*2*)
unfolding *nil eq-wset-def*
by (*metis* (*mono-tags*, *lifting*) *filter.simps*(*1*) *fold-Some-back fold-elem-back' fold-simps*(*1*) *list.collapse prod.exhaust-sel*)
show *?case*
unfolding *nil l'-def*
by *simp*
next
case *app*
obtain *a1 a2 l1* **where** *l-def*: $l = (a1,\ a2)\ \#\ l1$
using *app*

```
      by (metis neq-Nil-conv old.prod.exhaust)
     have A1': a ≠ a1 ⟹ (λx. ¬ (case x of (a', uu-) ⇒ a1 = a') ∧ (case x of
(a', uu-) ⇒ a = a')) = (λ(a', -). a = a') for a
      by (rule ext) force
    have A1: eq-wset (filter (λx. ¬ (case x of (a', uu-) ⇒ a1 = a')) l) (filter (λx.
¬ (case x of (a', uu-) ⇒ a1 = a')) l')
      using l-l'-eq
      unfolding eq-wset-def filter-filter
     proof (safe)
      fix a
      assume H: ∀ q. sum-key l q = sum-key l' q
      show fold (λ(-, w). (+) (Some w)) (filter (λx. ¬ (case x of (a', -) ⇒ a1 =
a') ∧ (case x of (a', -) ⇒ a = a')) l) None =
           fold (λ(-, w). (+) (Some w)) (filter (λx. ¬ (case x of (a', -) ⇒ a1 =
a') ∧ (case x of (a', -) ⇒ a = a')) l') None
        using H[rule-format, of a]
        by (cases a = a1; simp add: A1'[of a])
     qed
     have A2: length (map fst (filter (λx. ¬ (case x of (a', uu-) ⇒ a1 = a')) l))
< length (map fst l)
      unfolding l-def length-map list.size add-Suc-right add-0-right less-Suc-eq-le
      by simp
     show ?case
      using l-l'-eq A1[THEN ind[of filter (λx. ¬ (case x of (a', uu-) ⇒ a1 = a'))
l filter (λx. ¬ (case x of (a', uu-) ⇒ a1 = a')) l', OF A2]]
       unfolding H2[of x f l λ(a', -). a1 = a'] H2[of x f l' λ(a', -). a1 = a']
eq-wset-def H3
      by (cases f a1 = x) simp-all
   qed
  qed
  then show ?thesis
   using A
   by blast
qed

lemma wimage-transfer[transfer-rule]: rel-fun (=) (rel-fun (pcr-wset (=) (=)) (pcr-wset
(=) (=)))
    (λf M b. Finite-Set.fold (λx. (+) (M x)) None {a. M a ≠ None ∧ f a = b})
wimage
proof −
  have H: Finite-Set.fold (λx. (+) (weight ws x)) None {a. (∃ y. weight ws a =
Some y) ∧ f a = x} = weight (abs-wset (map (map-prod f id) (rep-wset ws))) x
for f :: 'a ⇒ 'b and ws :: ('a, 'w :: ref-ab-semigroup-add) wset and x :: 'b
  proof (induction ws rule: wset.abs-induct)
   fix l :: ('a × 'w :: ref-ab-semigroup-add) list
   have Finite-Set.fold (λx. (+) (sum-key l x)) w
        {a. (∃ v. sum-key l a = Some v) ∧ f a = x} =
        fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). x = a') (map (map-prod f
id) l)) w for w
```

proof (*induction l arbitrary*: *w*)
fix *w* :: *'w* :: *ref-ab-semigroup-add option*
show *Finite-Set.fold* (λ*x*. (+) (*sum-key* [] *x*)) *w* {*b*. (∃ *v*. *sum-key* [] *b* = *Some v*) ∧ *f b* = *x*} =
fold (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *x* = *a'*) (*map* (*map-prod f id*) [])) *w*
by *simp*
next
fix *a* :: *'a* × *'w*
and *l* :: (*'a* × *'w* :: *ref-ab-semigroup-add*) *list*
and *w* :: *'w* :: *ref-ab-semigroup-add option*
obtain *a1 a2* **where** *a-def*: *a* = (*a1*, *a2*)
by *fastforce*
assume *ind*: ⋀*w*. *Finite-Set.fold* (λ*x*. (+) (*sum-key l x*)) *w* {*a*. (∃ *v*. *sum-key l a* = *Some v*) ∧ *f a* = *x*} =
fold (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *x* = *a'*) (*map* (*map-prod f id*) *l*)) *w*
consider *f a1* = *x* | *f a1* ≠ *x*
by *fast*
then show *Finite-Set.fold* (λ*x*. (+) (*sum-key* (*a* # *l*) *x*)) *w* {*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*} = *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a'*, -). *x* = *a'*) (*map* (*map-prod f id*) (*a* # *l*))) *w*
proof (*cases*, *goal-cases eq neq*)
case *eq*
have *A1'*: {*b*. (∃ *v*. *sum-key l b* = *Some v*) ∧ *f b* = *x*} ⊆ *set* (*map fst l*) **for** *l* :: (*'a* × *'w* :: *ref-ab-semigroup-add*) *list*
by (*induction l*) *auto*
have *cfco-al*: *comp-fun-commute-on UNIV* (λ*x*. (+) (*sum-key* (*a* # *l*) *x*))
unfolding *comp-fun-commute-on-def comp-def*
by (*simp add*: *add.left-commute*)
have *fin-al*: *finite* {*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*}
by (*rule rev-finite-subset*[*of set* (*map fst* (*a* # *l*))])
(*simp*, *unfold a-def*, *use eq A1'* **in** *blast*)
have *mem-al*: *a1* ∈ {*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*}
by(*cases sum-key l a1*; *simp add*: *a-def eq fold-Some-out*)
have *A1*: *Finite-Set.fold* (λ*x*. (+) (*sum-key* (*a* # *l*) *x*)) *w*
{*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*} =
(*sum-key* (*a* # *l*) *a1*) +
Finite-Set.fold (λ*x*. (+) (*sum-key* (*a* # *l*) *x*)) *w*
({*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*} − {*a1*})
by (*rule comp-fun-commute-on.fold-rec*[*OF cfco-al subset-UNIV fin-al mem-al*])
have *A2*: *Finite-Set.fold* (λ*x*. (+) (*sum-key* (*a* # *l*) *x*)) *w* ({*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*} − {*a1*}) =
Finite-Set.fold (λ*x*. (+) (*sum-key l x*)) *w* ({*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*} − {*a1*})
by(*rule fold-closed-eq*[*of - UNIV*]; *simp add*: *eq a-def*)
consider *sum-key l a1* = *None* | *sum-key l a1* ≠ *None*
by *linarith*

then show *?case*
proof (*cases*, *goal-cases None Some*)
case *None*
have *A3*: {*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*} − {*a1*} = {*b*. (∃ *v*. *sum-key l b* = *Some v*) ∧ *f b* = *x*}
by(*auto simp add*: *None a-def eq fold-Some-out*)
show *?case*
unfolding *A1 A2*
unfolding *A3 ind*
by(*cases w*; *simp add*: *fold-Some-out a-def eq None add.commute add.assoc*[*symmetric*])
next
case *Some*
have *A3*: {*b*. (∃ *v*. *sum-key* (*a* # *l*) *b* = *Some v*) ∧ *f b* = *x*} − {*a1*} = {*b*. (∃ *v*. *sum-key l b* = *Some v*) ∧ *f b* = *x*} − {*a1*}
by(*auto simp add*: *Some a-def eq fold-Some-out*)
have *cfco-l*: *comp-fun-commute-on UNIV* (λ*x*. (+) (*sum-key l x*))
unfolding *comp-fun-commute-on-def comp-def*
by (*simp add*: *add.left-commute*)
have *fin-l*: *finite* {*b*. (∃ *v*. *sum-key l b* = *Some v*) ∧ *f b* = *x*}
by (*metis* (*lifting*) *A3 finite-insert finite.emptyI fin-al finite-Diff2*)
have *mem-l*: *a1* ∈ {*b*. (∃ *v*. *sum-key l b* = *Some v*) ∧ *f b* = *x*}
using *Some eq* **by** *blast*
have *A4*: *sum-key l a1* + *Finite-Set.fold* (λ*x*. (+) (*sum-key l x*)) *w* ({*b*. (∃ *v*. *sum-key l b* = *Some v*) ∧ *f b* = *x*} − {*a1*}) =
Finite-Set.fold (λ*x*. (+) (*sum-key l x*)) *w* {*b*. (∃ *v*. *sum-key l b* = *Some v*) ∧ *f b* = *x*}
by (*rule comp-fun-commute-on.fold-rec*[*symmetric*, *OF cfco-l subset-UNIV fin-l mem-l*])
have *A5*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a*′, -). *a1* = *a*′) (*a* # *l*)) *None* +
Finite-Set.fold (λ*x*. (+) (*fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a*′, -). *x* = *a*′) *l*) *None*)) *w*
({*b*. (∃ *v*. *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a*′, -). *b* = *a*′) *l*) *None* = *Some v*) ∧ *f b* = *x*} − {*a1*}) =
Some a2 + (*fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a*′, -). *a1* = *a*′) *l*) *None* +
Finite-Set.fold (λ*x*. (+) (*fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a*′, -). *x* = *a*′) *l*) *None*)) *w*
({*b*. (∃ *v*. *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a*′, -). *b* = *a*′) *l*) *None* = *Some v*) ∧ *f b* = *x*} − {*a1*}))
by(*simp add*: *add.commute add.assoc*[*symmetric*] *fold-Some-out a-def*)
show *?case*
unfolding *A1 A2*
unfolding *A3 A4 A5 ind*
by(*cases w*; *simp add*: *add.commute add.assoc*[*symmetric*] *fold-Some-out a-def eq*)
qed
next

```
      case neq
      have set-eq: {b. (∃ v. sum-key (a # l) b = Some v) ∧ f b = x} = {b. (∃ v.
sum-key l b = Some v) ∧ f b = x}
        using neq a-def
        by force
      from neq have Fold-eq: Finite-Set.fold (λx. (+) (sum-key (a # l) x)) w {a.
(∃ v. sum-key l a = Some v) ∧ f a = x} =
                        Finite-Set.fold (λx. (+) (sum-key l x)) w {a. (∃ v. sum-key l a
= Some v) ∧ f a = x}
        by (intro fold-closed-eq[of - UNIV]) (auto simp: a-def)
      show ?case
        unfolding set-eq Fold-eq
        using ind[of w] a-def neq
        by simp
    qed
  qed
   then show Finite-Set.fold (λx. (+) (weight (abs-wset l) x)) None {a. (∃ v.
weight (abs-wset l) a = Some v) ∧ f a = x} = weight (abs-wset (map (map-prod f
id) (rep-wset (abs-wset l)))) x
   unfolding weight.abs-eq fold-eq-wset[OF eq-wset-refl[THEN Weighted-Set.Quotient-wset[THEN
Quotient-rep-abs], of l], symmetric]
    by fast
 qed
 show ?thesis
   unfolding wimage-def pcr-wset-def cr-wset-def option.rel-eq fun.rel-eq eq-OO
map-fun-def comp-def
  unfolding rel-fun-def
  by(simp add: H)
qed

lemma wfilter-transfer[transfer-rule]: rel-fun (rel-fun (=) (=)) (rel-fun (pcr-wset
(=) (=)) (pcr-wset (=) (=)))
    (λf M b. case (M b, f b) of (Some b', True) ⇒ Some b' | - ⇒ None) wfilter
proof −
 have ( case fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). x = a') xs) v of None
⇒ None
    | Some b' ⇒ (case ws x of True ⇒ Some b' | False ⇒ None)) =
   fold (λ(a, w). (+) (Some w))
    (filter (λaw. ws (fst aw) ∧ (case aw of (a', -) ⇒ x = a')) xs) v
   if v ≠ None ⟶ ws x
  for x :: 'a and ws :: 'a ⇒ bool and ws' :: 'a ⇒ bool and v :: 'w :: ref-ab-semigroup-add
option and xs :: ('a × 'w :: ref-ab-semigroup-add) list and c :: bool
 proof (cases ws x)
   case True
   have filter-eq: (filter (λaw. ws (fst aw) ∧ (case aw of (a', -) ⇒ x = a')) xs) =
                (filter (λ(a', -). x = a') xs)
     by (rule filter-cong) (auto simp: True case-prod-beta)
   show ?thesis by (simp add: True filter-eq split: option.splits)
 next
```

case *False*
then have *v-None*: $v = None$ **using** *that* **by** *simp*
have *filter-empty*: *filter* $(\lambda aw.\ ws\ (fst\ aw) \wedge (case\ aw\ of\ (a',\ \text{-}) \Rightarrow x = a'))\ xs = []$
by (*rule filter-False*) (*auto simp add*: *False case-prod-beta*)
have *lhs-None*: (*case fold* $(\lambda(a,\ w).\ (+)\ (Some\ w))$ (*filter* $(\lambda(a',\ \text{-}).\ x = a')\ xs)\ v$ *of*
None ⇒ *None* | *Some* b' ⇒ *if ws x then Some* b' *else None*) = *None*
proof (*cases filter* $(\lambda(a',\ \text{-}).\ x = a')\ xs$)
case *Nil*
then show *?thesis* **by** (*simp add*: *v-None*)
next
case (*Cons h t*)
then have *fold* $(\lambda(a,\ w).\ (+)\ (Some\ w))$ (*filter* $(\lambda(a',\ \text{-}).\ x = a')\ xs)\ v \neq None$
unfolding *v-None* **using** *fold-some'* **by** *metis*
then obtain b' **where** *fold* $(\lambda(a,\ w).\ (+)\ (Some\ w))$ (*filter* $(\lambda(a',\ \text{-}).\ x = a')\ xs)\ v = Some\ b'$
by *blast*
then show *?thesis* **by** (*simp add*: *False*)
qed
show *?thesis* **by** (*simp add*: *filter-empty lhs-None v-None False split*: *option.splits bool.splits*)
qed
then show *?thesis* **unfolding** *rel-fun-def pcr-wset-def cr-wset-def option.rel-eq relcompp-apply wfilter-def*
weight.abs-eq map-fun-def o-apply
by (*auto simp*: *weight.rep-eq*)
qed

end

lemma *wimage-empty*[*simp*]: *wimage f wempty* = *wempty*
by (*transfer*) (*simp add*: *eq-wset-def*)

lemma *wimage-wadd-wsingle*: *wimage f* (*wadd* (*wsingle x w*) *M*) = *wadd* (*wsingle* (*f x*) *w*) (*wimage f M*)
by (*transfer*) (*auto simp add*: *eq-wset-def*)

lemma *wimage-wsingle*[*simp*]: *wimage f* (*wsingle x w*) = (*wsingle* (*f x*) *w*)
by (*transfer*) (*auto simp add*: *eq-wset-def*)

lemma *wimage-wadd*[*simp*]: *wimage f* (*wadd xs ys*) = *wadd* (*wimage f xs*) (*wimage f ys*)
by (*transfer*) *auto*

lemma *w-image-update*:
weight M x = *None* ⟹ *wimage f* (*wupdate M x* (*Some w*)) = *wadd* (*wsingle* (*f*

x) *w*) (*wimage f M*)
proof(*transfer*, *safe*)
fix *M* :: (*'b* × (*'a* :: *ref-ab-semigroup-add*)) *list*
and *x* :: *'b*
and *f* :: *'b* ⇒ *'c*
and *w* :: *'a*
assume *sum-key M x* = *None*
then have *filter-M-eq*: *filter* (λ(*x'*, -). *x* ≠ *x'*) *M* = *M*
proof (*induction M*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons h t*)
then show *?case* **by** (*auto simp add*: *fold-Some*)
qed
show *eq-wset*
(*map* (*map-prod f id*)
(*case Some w of None* ⇒ *filter* (λ(*x'*, -). *x* ≠ *x'*) *M*
| *Some w'* ⇒ (*x*, *w'*) # *filter* (λ(*x'*, -). *x* ≠ *x'*) *M*))
([(*f x*, *w*)] @ *map* (*map-prod f id*) *M*)
by(*simp add*: *filter-M-eq*)
qed

lifting-update *wset.lifting*
lifting-forget *wset.lifting*

locale *Quotient-wset* **begin**
setup-lifting *Weighted-Set.Quotient-wset Weighted-Set.wset-equivp*[*THEN equivp-reflp2*]
end

lemma *abs-wset-rep-wset*: *abs-wset* (*rep-wset x*) = *x*
by (*rule Quotient-abs-rep*[*OF Quotient-wset*])
lemma *abs-wset-cons*: *abs-wset* ((*x*,*w*) # *xs*) = *wadd* (*wsingle x w*) (*abs-wset xs*)
by *transfer auto*

lemma *abs-wset-map*: *abs-wset* (*map* (*map-prod f id*) *xs*) = *wimage f* (*abs-wset xs*)
by *transfer auto*

context begin
interpretation *wset-as-pfun*.

lemma *rep-wset-set*:
assumes (*a*, *w*) ∈ *set* (*rep-wset z*)
shows ∃ *y*. *weight z a* = *Some y*
proof −
have *H*: (*a*, *w*) ∈ *set xs* ⟹ ∃ *y*. *sum-key xs a* = *Some y* **for** *a* :: *'a* **and** *w* :: *'w* :: *ref-ab-semigroup-add* **and** *xs* :: (*'a* × *'w* :: *ref-ab-semigroup-add*) *list*
proof (*induction xs*)

```
    case Nil then show ?case by simp
  next
    case (Cons x xs)
    then show ?case by (cases x; auto simp add: fold-Some')
  qed
  then show ?thesis
    using assms
    unfolding weight-def
    by(simp add: H)
qed

lemma set-wset-in: x ∈ set-wset ws = (weight (ws :: ('a, 'w :: ref-ab-semigroup-add)
wset) x ≠ None)
proof −
  have H1: filter (λ(a', -). e = a') xs ≠ [] ⟹ sum-key (map (map-prod Inr id)
xs) (Inr e) ≠ None for e :: 'a and xs :: ('a × 'w :: ref-ab-semigroup-add) list
  proof (induction xs)
    case Nil then show ?case by simp
  next
    case (Cons x xs)
    then show ?case by (cases x; cases e = fst x; auto simp add: fold-Some)
  qed
  have key: (∀ zs. eq-wset (map (map-prod Inr id) (rep-wset ws)) zs ⟶
            (∃ zs∈set zs. ∃ zs∈Basic-BNFs.fsts zs. x ∈ Basic-BNFs.setr zs)) =
           (∃ y. weight ws x = Some y)
  proof (rule iffI)
    assume H: ∀ zs. eq-wset (map (map-prod Inr id) (rep-wset ws)) zs ⟶
            (∃ z∈set zs. ∃ a∈Basic-BNFs.fsts z. x ∈ Basic-BNFs.setr a)
    have mem: ∃ aw∈set (rep-wset ws). x = fst aw
    proof −
      from H[THEN spec[where x = map (map-prod Inr id) (rep-wset ws)],
          simplified eq-wset-refl simp-thms]
      show ?thesis
      by (simp add: image-def Bex-def Basic-BNFs.fsts.simps Basic-BNFs.setr.simps)
    qed
    then obtain a w where (a, w) ∈ set (rep-wset ws) and x = a by auto
    then show ∃ y. weight ws x = Some y by (blast dest: rep-wset-set)
  next
    assume H: ∃ y. weight ws x = Some y
    show ∀ zs. eq-wset (map (map-prod Inr id) (rep-wset ws)) zs ⟶
            (∃ zs∈set zs. ∃ zs∈Basic-BNFs.fsts zs. x ∈ Basic-BNFs.setr zs)
    proof (rule allI, rule impI)
      fix zs :: (('d + 'a) × 'w) list
      assume Heq: eq-wset (map (map-prod Inr id) (rep-wset ws)) zs
      from H obtain y where Hweight: weight ws x = Some y by blast
      have sk-ne: sum-key (map (map-prod Inr id) (rep-wset ws)) (Inr x) ≠ None
      proof −
        from Hweight have weight ws x ≠ None by simp
        then show ?thesis
```

```
      unfolding weight-def
      by (cases filter (λ(a', -). x = a') (rep-wset ws)) (simp-all add: H1)
    qed
    have sk-zs: sum-key zs (Inr x) ≠ None
      using Heq sk-ne unfolding eq-wset-def by metis
    show ∃ zs∈set zs. ∃ zs∈Basic-BNFs.fsts zs. x ∈ Basic-BNFs.setr zs
    proof −
      from sk-zs obtain v where sum-key zs (Inr x) = Some v by fastforce
      then have Inr x ∈ fst ‘ set zs by (simp add: in-set-conv-sum-key-Some)
      then obtain p where hp: p ∈ set zs and fst p = Inr x by auto
      then show ?thesis
        by (auto simp add: Basic-BNFs.fsts.simps Basic-BNFs.setr.simps intro:
bexI[OF - hp])
    qed
  qed
 qed
 then show ?thesis unfolding set-wset-def by simp
qed



lemma set-wset-alt-def: set-wset ws = {x. weight ws x ≠ None}
 using set-wset-in
 by fast

lemma wrel-alt-def:
 fixes x :: ('a, 'w :: ref-ab-semigroup-add) wset and y :: ('b, 'w) wset
 shows wrel R x y = (∃ xs ys. abs-wset xs = x ∧ list-all2 (rel-prod R (=)) xs ys
∧ abs-wset ys = y)
 unfolding wset.in-rel set-wset-alt-def
proof safe
 fix z :: ('a × 'b, 'w) wset
 assume {x. weight z x ≠ None} ⊆ {(x, y). R x y}
 then have list-all2 (rel-prod R (=)) (map (λ((a,b),c). (a,c)) (rep-wset z)) (map
(λ((a,b),c). (b,c)) (rep-wset z))
   unfolding list.rel-map by (auto intro!: list.rel-refl-strong simp: subset-eq dest:
rep-wset-set)
 moreover have abs-wset (map (λ((a,b),c). (a,c)) (rep-wset z)) = wimage fst z
  by (metis (no-types, lifting) abs-wset-map abs-wset-rep-wset id-apply map-eq-conv
map-prod-def split-beta)
 moreover have abs-wset (map (λ((a,b),c). (b,c)) (rep-wset z)) = wimage snd z
  by (metis (no-types, lifting) abs-wset-map abs-wset-rep-wset id-apply map-eq-conv
map-prod-def split-beta)
 ultimately show ∃ xs ys. abs-wset xs = wimage fst z ∧ list-all2 (rel-prod R (=))
xs ys ∧ abs-wset ys = wimage snd z
   by blast
next
 fix xs :: ('a × 'w) list and ys :: ('b × 'w) list
 assume list-all2 (rel-prod R (=)) xs ys
```

```
  then obtain zs :: (('a × 'w) × ('b × 'w)) list where zs: map fst zs = xs map
snd zs = ys
    set zs ⊆ {(x, y). rel-prod R (=) x y}
    unfolding list.in-rel by blast
  let ?zs = abs-wset (map (λ((a, -), (b, w)). ((a, b), w)) zs)
  from zs(3) have ?zs ∈ {x. {a. weight x a ≠ None} ⊆ {(x, y). R x y}}
    by (induct zs) (auto simp: weight.abs-eq)
  moreover from zs(1,3) have wimage fst ?zs = abs-wset xs
    by (force simp flip: abs-wset-map simp: wset.abs-eq-iff eq-wset-def
        intro!: arg-cong2[where f = sum-key])
  moreover from zs(2) have wimage snd ?zs = abs-wset ys
    by (force simp flip: abs-wset-map simp: wset.abs-eq-iff eq-wset-def
        intro!: arg-cong2[where f = sum-key])
  ultimately show ∃ z. z ∈ {M. {xy. weight M xy ≠ None} ⊆ {(x, y). R x y}} ∧
    wimage fst z = abs-wset xs ∧ wimage snd z = abs-wset ys
    by blast
qed

end

declare [[typedef-overloaded]]
codatatype ('a, 'w) wsetinf = WSetInf (('a, 'w) wsetinf + 'a, 'w :: ref-ab-semigroup-add)
wset

end
```

References

[1] H. P. Gumm and T. Schröder. Monoid-labeled transition systems. In A. Corradini, M. Lenisa, and U. Montanari, editors, *CMCS 2001*, volume 44 of *Electronic Notes in Theoretical Computer Science*, pages 185–204. Elsevier, 2001.