

Formalization of Weighted Sets

Mathias Schack Rabing Dmitriy Traytel

June 17, 2026

Abstract

We define *weighted sets* as a generalization of finite multisets where multiplicities are replaced by weights from a *refinable* [1] *abelian semi-group*. 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

1 Algebraic Preliminaries	2
2 The Positive Representation	3
3 Basic Operations	7
4 The Splitting Relation	10
5 The Negative Representation	40
6 BNF Registration	46
7 Further Operations	47
8 Switching Between Representations	48

```

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)
 $\Rightarrow$  Some (x + y) | (Some x, None)  $\Rightarrow$  Some x | (None, Some x)  $\Rightarrow$  Some x | -  $\Rightarrow$ 
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  $\implies$ 
    ( $\exists$  (e11 :: 'a :: ab-semigroup-add) option) e12 e21 e22. Some a = e11 + e12  $\wedge$ 
      Some b = e21 + e22  $\wedge$  Some c = e11 + e21  $\wedge$  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'  $\Rightarrow$  e + f' | None
 $\Rightarrow$  e) f + Some e = Some (case f of Some f'  $\Rightarrow$  f' + e | None  $\Rightarrow$  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  $\exists$  e11 e12 e21 e22. Some a = e11 + e12  $\wedge$  Some b = e21 + e22  $\wedge$  Some
c = e11 + e21  $\wedge$  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
end

```

2 The Positive Representation

abbreviation *sum-key* where

sum-key $kxs\ e \equiv \text{fold } (\lambda(-,w)\ y.\ \text{Some } w + y)\ (\text{filter } (\lambda(e',-).\ e = e')\ kxs)\ \text{None}$

definition *eq-wset* where

eq-wset $(kxs :: ('a \times ('w :: \text{ref-ab-semigroup-add}))\ \text{list})\ (kys :: ('a \times ('w :: \text{ref-ab-semigroup-add}))\ \text{list}) =$

$(\forall\ e.\ \text{sum-key } kxs\ e = \text{sum-key } kys\ e)$

declare $[[\text{typedef-overloaded}]]$

quotient-type $('a,\ 'w)\ \text{wset} = ('a \times 'w :: \text{ref-ab-semigroup-add})\ \text{list} / \text{eq-wset}$
by $(\text{auto intro!:\ equivI reflpI sympI transpI simp:\ eq-wset-def})$

lemma *get-abs-wset*: $\exists l.\ M = \text{abs-wset } l$

by $(\text{metis Quotient3-abs-rep Quotient3-wset})$

lemma *fold-Some*: $\text{fold } (\lambda(a, w).\ (+)\ (\text{Some } w))\ xs\ (\text{Some } w) \neq \text{None}\ \text{None} \neq$
 $\text{fold } (\lambda(a, w).\ (+)\ (\text{Some } w))\ xs\ (\text{Some } w)$

proof–

have $H:\ \text{fold } (\lambda(a, w).\ (+)\ (\text{Some } w))\ xs\ (\text{Some } w) \neq \text{None}$

proof $(\text{induction } xs\ \text{arbitrary: } w)$

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 $\forall a. (a \notin \text{set} (\text{map fst } xs)) = (a \notin \text{set} (\text{map fst } xs'))$
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' \implies eq-wset xs' xs*
unfolding *eq-wset-def*
by *fastforce*

lemma *eq-wset-trans*: *eq-wset xs ys \implies eq-wset ys zs \implies 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 ($\lambda(a, w). (+) (Some w)$) (filter ($\lambda(a', -). a = a'$) (xs @ e1 # e2 # x's)) w =*
fold ($\lambda(a, w). (+) (Some w)$) (filter ($\lambda(a', -). a = a'$) (xs @ e2 # e1 # x's)) w
for *a :: 'c and e1 e2 :: 'c \times ('d :: ab-semigroup-add) and xs x's :: ('c \times 'd) list*
and *w :: 'd option*
by(*induction xs; (auto simp add: add.assoc[symmetric], simp only: add.commute)*)

lemma *fold-elem-back*: *fold ($\lambda(a, w). (+) (Some w)$) (filter ($\lambda(a', -). a = a'$) (xs @ e # x's)) w =*
fold ($\lambda(a, w). (+) (Some w)$) (filter ($\lambda(a', -). a = a'$) (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'*: $\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a') (e \# x's))\ w =$
 $\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a') (x's\ @\ [e]))\ w$
by (*metis append-self-conv2 fold-elem-back*)

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

lemma *fold-Some-back*: $\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a')\ xs)$
 $(Some\ (M :: - :: \text{ref-ab-semigroup-add})) = \text{sum-key } (xs\ @\ [(a, M)])\ a$

proof –

have *fold-Some-front*: $\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a')\ xs)$
 $(Some\ M) = \text{sum-key } ((a, M) \# xs)\ a$

by *simp*

show *?thesis*

using *eq-wset-elem-back'*

unfolding *eq-wset-def fold-Some-front*

by *metis*

qed

lemma *fold-back'*: $\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a')\ xs)\ (Some$
 $(M :: - :: \text{ref-ab-semigroup-add}) + w) = \text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a',$
 $-). a = a')\ (xs\ @\ [(a, M)]))\ w$

proof –

have *fold-Some-front'*: $\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a')\ xs)$
 $(Some\ M + w) = \text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a')\ ((a, M) \#$
 $xs))\ w$

by *simp*

show *?thesis*

using *fold-elem-back'*

unfolding *eq-wset-def fold-Some-front'*

by *metis*

qed

lemma *eq-wset-append*: $\text{eq-wset } xs\ xs' \implies \text{eq-wset } ys\ ys' \implies \text{eq-wset } (xs\ @\ ys)$
 $(xs' @ ys')$

proof –

assume *H1*: *eq-wset xs xs'* and *H2*: *eq-wset ys ys'*

have *xs-eq*: $\forall a. \text{sum-key } xs\ a = \text{sum-key } xs'\ a$

using *H1* unfolding *eq-wset-def* by *simp*

have *ys-eq*: $\forall a. \text{sum-key } ys\ a = \text{sum-key } ys'\ a$

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*: $\text{sum-key } xs\ a = \text{sum-key } xs'\ a$ 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'  $\implies$  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  $\times$  'b
  and xs :: ('a  $\times$  '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])  $\implies$  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 ::  $\langle ('a, 'w :: \text{ref-ab-semigroup-add}) \text{wset} \rangle$  is
 $\langle [] \rangle$  .

```

```

lift-definition weight ::  $\langle ('a, 'w :: \text{ref-ab-semigroup-add}) \text{wset} \Rightarrow 'a \Rightarrow 'w \text{option} \rangle$ 
is

```

```

 $\langle \lambda kxs e. \text{sum-key } kxs e \rangle$ 

```

```

unfolding eq-wset-def

```

```

by simp

```

```

lemma weight-eq-iff:  $(\forall x. \text{weight } M x = \text{weight } N x) = (M = N)$ 
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* :: $\langle 'a \Rightarrow 'w \Rightarrow ('a, 'w :: \text{ref-ab-semigroup-add}) \text{wset} \rangle$ **is**
 $\langle \lambda a w. [(a,w)] \rangle$.

lift-definition *wset* :: $\langle ('a, 'w :: \text{ref-ab-semigroup-add}) \text{wset} \Rightarrow 'a \text{ set} \rangle$ **is**
 $\langle \lambda M. \text{set} (\text{map fst } M) \rangle$ **by** (*drule eq-wset-fst*)(*auto simp add: image-def*)

lift-definition *wadd* :: $\langle ('a, 'w :: \text{ref-ab-semigroup-add}) \text{wset} \Rightarrow ('a, 'w) \text{wset} \Rightarrow ('a, 'w) \text{wset} \rangle$ **is**
 $\langle \lambda M1 M2. M1 @ M2 \rangle$ **using** *eq-wset-append* **by** *metis*

lemma *sum-key-wupdate-same*:

sum-key (*case w of None* \Rightarrow *filter* $(\lambda(x', -). x \neq x')$ *l*
 $|$ *Some w'* \Rightarrow $(x, w') \#$ *filter* $(\lambda(x', -). x \neq x')$ *l*) $x = w$
for *l* :: $('a \times ('w :: \text{ref-ab-semigroup-add}))$ *list*
by (*cases w*) (*simp-all add: case-prod-beta*)

lemma *sum-key-wupdate-diff*:

$x \neq x' \Longrightarrow$
sum-key (*case w of None* \Rightarrow *filter* $(\lambda(x'', -). x \neq x'')$ *l*
 $|$ *Some w'* \Rightarrow $(x, w') \#$ *filter* $(\lambda(x'', -). x \neq x'')$ *l*) $x' = \text{sum-key } l \ x'$
for *l* :: $('a \times ('w :: \text{ref-ab-semigroup-add}))$ *list*

proof (*cases w*)

case *None*

assume *neq*: $x \neq x'$

have *filter-eq*: *filter* $(\lambda y. x \neq \text{fst } y \wedge x' = \text{fst } y)$ *l* = *filter* $(\lambda(e', -). x' = e')$ *l* \wedge
filter $(\lambda y. x' = \text{fst } y \wedge x \neq \text{fst } y)$ *l* = *filter* $(\lambda(e', -). x' = e')$ *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*: $x \neq x'$

have *filter-eq*: *filter* $(\lambda y. x \neq \text{fst } y \wedge x' = \text{fst } y)$ *l* = *filter* $(\lambda(e', -). x' = e')$ *l* \wedge
filter $(\lambda y. x' = \text{fst } y \wedge x \neq \text{fst } y)$ *l* = *filter* $(\lambda(e', -). x' = e')$ *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* :: $\langle ('a, 'w :: \text{ref-ab-semigroup-add}) \text{wset} \Rightarrow 'a \Rightarrow 'w \text{ option} \Rightarrow ('a, 'w) \text{wset} \rangle$ **is**

$\langle \lambda M x w. \text{case } w \text{ of } \text{Some } w' \Rightarrow (x, w') \# (\text{filter } (\lambda(x', -). x \neq x') M) \mid \text{None} \Rightarrow \text{filter } (\lambda(x', -). x \neq x') M \rangle$

proof –

fix *l1 l2* :: $('a \times ('w :: \text{ref-ab-semigroup-add}))$ *list* **and** *a* :: *'a* **and** *w* :: *'w option*

assume *hyp*: *eq-wset l1 l2*

show *eq-wset* (*case w of Some w'* \Rightarrow $(a, w') \#$ *filter* $(\lambda(x', -). a \neq x')$ *l1*

```

      | None  $\Rightarrow$  filter ( $\lambda(x', -). a \neq x'$ ) l1
    (case w of Some w'  $\Rightarrow$  (a, w') # filter ( $\lambda(x', -). a \neq x'$ ) l2
      | None  $\Rightarrow$  filter ( $\lambda(x', -). a \neq x'$ ) l2)
  unfolding eq-wset-def
proof
  fix a'
  show sum-key (case w of Some w'  $\Rightarrow$  (a, w') # filter ( $\lambda(x', -). a \neq x'$ ) l1
    | None  $\Rightarrow$  filter ( $\lambda(x', -). a \neq x'$ ) l1) a' =
    sum-key (case w of Some w'  $\Rightarrow$  (a, w') # filter ( $\lambda(x', -). a \neq x'$ ) l2
      | None  $\Rightarrow$  filter ( $\lambda(x', -). a \neq x'$ ) 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  $\times$  ('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 = ( $\lambda$ - . 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:  $x \neq x' \implies \text{weight } (wupdate\ M\ x\ w)\ x' = \text{weight } M\ x'$  for  $x'$ 
  by transfer (rule sum-key-wupdate-diff)
show ?thesis
proof (rule ext)
  fix  $x'$ 
  show  $\text{weight } (wupdate\ M\ x\ w)\ x' = ((\text{weight } M)(x := w))\ x'$ 
  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

$fold'\ l \equiv foldl\ (+)\ (hd\ l)\ (tl\ l)$

inductive *list-split* :: $('a \times 'w :: \text{ref-ab-semigroup-add})\ list \Rightarrow ('a \times 'w)\ list \Rightarrow \text{bool}$
where

Base: $list-split\ []\ []$

| *Split*: $list-split\ xs''\ ys \implies xs = xs' @ xs'' \implies w = fold'\ (map\ snd\ xs') \implies xs' \neq [] \implies list-all\ (\lambda(a,b). a = y)\ xs' \implies list-split\ xs\ ((y,w) \# ys)$

inductive *list-split'* :: $((('w :: \text{ref-ab-semigroup-add})\ option)\ list)\ list \Rightarrow ('w\ option)\ list \Rightarrow \text{bool}$ **where**

Base': $list-split'\ []\ []$

| *Split'*: $list-split'\ xss\ ys \implies y = fold'\ xs \implies xs \neq [] \implies list-split'\ (xs \# xss)\ (y \# ys)$

lemma *list-split-cons-eq*: $list-split\ xs1\ xs2 \implies list-split\ (x \# xs1)\ (x \# xs2)$

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 $w + w' = \text{fold}' (\text{map snd } [(x, w), (x, w')])$ **by** *simp*

show $[(x, w), (x, w')] \neq []$ **by** *simp*

show *list-all* $(\lambda(a, b). a = x)$ $[(x, w), (x, w')]$ **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*: $xs \neq [] \implies \text{list-all } (\lambda(a, b). a = x) xs \implies \text{eq-wset } ((x, w) \# xs) [(x, \text{foldl } (+) w (\text{map snd } xs))]$ **for** $x :: 'a$ **and** $xs :: ('a \times ('w :: \text{ref-ab-semigroup-add})) \text{list}$ **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.prem by simp
                    have a-y: a = y using Cons.prem 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
          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
          qed
  qed

```

lemma *list-split-app*: $list-split\ xs\ (ys\ @\ xs') \implies \exists\ xs'\ xs''.\ list-split\ xs'\ ys \wedge list-split\ xs''\ ys' \wedge xs = xs' @ xs''$

```

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.premis have split-y: list-split xs ((a, w) # (yss @ ys'))
    unfolding y-def by simp
  then obtain xs1 xs2 where
    xs-def: xs = xs1 @ xs2
    and split-tail: list-split xs2 (yss @ ys')
    and w-def: w = fold' (map snd xs1)
    and xs1-ne: xs1 ≠ []
    and xs1-all: list-all (λ(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' @ xs''
  by blast
  have split-left: list-split (xs1 @ 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 (λ(a, b). a = e) xs ⇒ ∀ e'. filter (λ(a', -). e' = a') xs = (if
e = e' then xs else []) for e :: 'a and xs :: ('a × '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 (λ(a, w). (+) (Some w)) (if e = fst y then y # filter (λ(a',
-). e = a') ys else filter (λ(a', -). e = a') ys) (ws e) =
    fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). e = a') ys) ((ws(fst y:= Some
(snd y) + ws (fst y))) e) for e :: 'a and ws :: 'a ⇒ ('w :: ref-ab-semigroup-add)
option and ys :: ('a × 'w) list and y :: ('a × '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 (λ(a, b). a = e) xs ⇒
    ∀ a. fold (λ(a, w). (+) (Some w)) (if e = a then xs else []) None = fold (λ(a,

```

$w). (+) (Some\ w)) (filter\ (\lambda(a',\ -).\ a = a')\ ys)\ (w\ a) \implies list\ all\ (\lambda(a,\ b).\ a = e)$
 $ys\ for\ e :: 'a\ and\ xs :: ('a \times ('w :: ref\ ab\ semigroup\ add))\ list\ and\ ys :: ('a \times 'w)$
 $list\ and\ w :: 'a \implies 'w\ option$
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 ($\lambda(a,\ b).\ a = e$) xs
assume hall: $\forall a.$ fold ($\lambda(a,\ w). (+) (Some\ w)$) (if $e = a$ then xs else []) None
=

$fold\ (\lambda(a,\ w). (+) (Some\ w)) (filter\ (\lambda(a',\ -).\ a = a')\ (y\ \# ys'))$
(w a)

have hall': $\forall a.$ fold ($\lambda(a,\ w). (+) (Some\ w)$) (if $e = a$ then xs else []) None =
 $fold\ (\lambda(a,\ w). (+) (Some\ w)) (filter\ (\lambda(a',\ -).\ a = a')\ ys')$
 $((w(fst\ y := Some\ (snd\ y) + w\ (fst\ y)))\ a)$
using hall by (simp add: fold-upd[of - - w] case-prod-beta)
have ys'-all: list-all ($\lambda(a,\ b).\ a = e$) 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: $fst\ y \neq e$
have h-fy: fold ($\lambda(a,\ w). (+) (Some\ w)$) (if $e = fst\ y$ then xs else []) None =
 $fold\ (\lambda(a,\ w). (+) (Some\ w)) (filter\ (\lambda(a',\ -).\ fst\ y = a')\ (y\ \# ys'))$
(w (fst y))

using hall[THEN spec, of fst y] by simp
have lhs: fold ($\lambda(a,\ w). (+) (Some\ w)$) (if $e = fst\ y$ then xs else []) None =
None

using ne by simp
have rhs-pos: fold ($\lambda(a,\ w). (+) (Some\ w)$) (filter ($\lambda(a',\ -).\ fst\ y = a')$ (y #
ys')) (w (fst y)) \neq 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 ($\lambda(a,\ b).\ a = e$) (y # ys')
using y-eq ys'-all by (cases y) auto
qed
have list-all-eq-wset: list-all ($\lambda(a,\ b).\ a = e$) xs $\implies eq\ wset\ xs\ ys \implies list\ all$
 $(\lambda(a,\ b).\ a = e)\ ys\ for\ e :: 'a\ and\ xs :: ('a \times ('w :: ref\ ab\ semigroup\ add))\ list$
and ys :: ('a \times 'w) list
unfolding eq-wset-def
by(simp add: H2' list-all-eq-aux[of e xs ys $\lambda.$ None])
have fold-Some-eq: fold ($\lambda(a,\ w). (+) (Some\ w)$) ys (Some wy) = fold ($\lambda(a,\ w).$
 $(+) (Some\ w)$) xs (Some wx) \implies
 $fold\ (\lambda x\ s.\ s + x)\ (map\ snd\ xs)\ wx = fold\ (\lambda x\ s.\ s + x)\ (map\ snd\ ys)$
wy **for** xs :: ('a \times ('w :: ref-ab-semigroup-add)) list **and** ys :: ('a \times '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 ( $\lambda(a, b). a = e$ ) xs
assume eq: eq-wset ys xs
have all-ys: list-all ( $\lambda(a, b). a = e$ ) ys
  using list-all-eq-wset[OF all-xs, OF eq-wset-sym[OF eq]] .
have ys-ne: ys  $\neq$  []
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  $\neq$  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 ( $\lambda(a, w). (+) (Some w)$ ) xs None
    by (simp add: H2'[OF all-xs])
  have ys-e: sum-key ys e = fold ( $\lambda(a, w). (+) (Some w)$ ) ys None
    by (simp add: H2'[OF all-ys])
  have key: fold ( $\lambda(a, w). (+) (Some w)$ ) ys None = fold ( $\lambda(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)  $\wedge$  ys  $\neq$  []  $\wedge$  list-all ( $\lambda(a, b). a =$ 
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
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  $\implies$  l  $\neq$  []  $\implies$  (option-list l)  $\neq$  []
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  $\implies$  l  $\neq$  []  $\implies$  a = fold' (option-list l)
proof -
  have H: Some x = foldl (+) (Some x') l  $\implies$  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  $\implies$  l  $\neq$  []  $\implies$  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  $\times$  'w) list  $\Rightarrow$  ('a  $\Rightarrow$  (('w option) list) list)  $\Rightarrow$  ('a  $\times$  'w) list
where
  create-split [] als = [] |
  create-split ((a,-) # l) als = map ( $\lambda$ x. (a,x)) (option-list (hd (als a))) @
  (create-split l (als(a := tl(als a))))

```

lemma list-split'-exist:

```

((xs  $\neq$  []  $\wedge$  ys  $\neq$  []))  $\longrightarrow$  fold' (xs :: (('w :: ref-ab-semigroup-add) option) list) =
fold' ys  $\implies$ 
  ((xs = []) = (ys = []))  $\implies$ 
  ( $\exists$  zs zs'. list-split' zs xs  $\wedge$  list-split' zs' ys  $\wedge$  ( $\forall$  n m. n < length 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 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:  $\forall$  m < length (xs :: (('w :: ref-ab-semigroup-add) option) list) + length
ys.

```

```

   $\forall$  (x :: (('w :: ref-ab-semigroup-add) option) list) ls'.
    m = length x + length ls'  $\longrightarrow$ 
    (x  $\neq$  []  $\wedge$  ls'  $\neq$  []  $\longrightarrow$  fold' x = fold' ls')  $\longrightarrow$ 
    (x = [] = (ls' = []))  $\longrightarrow$ 
    ( $\exists$  zs zs'.
      list-split' zs x  $\wedge$  list-split' zs' ls'  $\wedge$  ( $\forall$  n m. n < length x  $\longrightarrow$  m <
length ls'  $\longrightarrow$  zs ! n ! m = zs' ! m ! n)  $\wedge$  list-all ( $\lambda$ l. length l = length ls') zs  $\wedge$ 
list-all ( $\lambda$ l. length l = length x) zs')
  and eq-app: (xs  $\neq$  []  $\wedge$  ys  $\neq$  [])  $\longrightarrow$  fold' (xs :: (('w :: ref-ab-semigroup-add)
option) list) = fold' ys
  and eq-nil: (xs = []) = (ys = [])

```

have *ind*: $\bigwedge(x :: (('w :: \text{ref-ab-semigroup-add}) \text{option}) \text{list}) \text{ls}'$
 $\text{length } x + \text{length } \text{ls}' < \text{length } xs + \text{length } ys \implies$
 $(x \neq [] \wedge \text{ls}' \neq [] \longrightarrow \text{fold}' x = \text{fold}' \text{ls}') \implies$
 $(x = [] = (\text{ls}' = [])) \implies$
 $(\exists zs \text{zs}'.$
 $\text{list-split}' zs x \wedge \text{list-split}' \text{zs}' \text{ls}' \wedge (\forall n m. n < \text{length } x \longrightarrow m <$
 $\text{length } \text{ls}' \longrightarrow zs ! n ! m = \text{zs}' ! m ! n) \wedge \text{list-all } (\lambda l. \text{length } l = \text{length } \text{ls}') zs \wedge$
 $\text{list-all } (\lambda l. \text{length } l = \text{length } x) \text{zs}') \implies$
using *ind by auto*
have *gt3-cases*: $\bigwedge(xs :: 'w \text{option list}) ys. ((xs = []) = (ys = [])) \implies$
 $((xs \neq [] \wedge ys \neq []) \longrightarrow \text{fold}' (xs :: (('w :: \text{ref-ab-semigroup-add}) \text{option})$
 $\text{list}) = \text{fold}' ys) \implies$
 $(\bigwedge(x :: (('w :: \text{ref-ab-semigroup-add}) \text{option}) \text{list}) \text{ls}'$
 $\text{length } x + \text{length } \text{ls}' < \text{length } xs + \text{length } ys \implies$
 $(x \neq [] \wedge \text{ls}' \neq [] \longrightarrow \text{fold}' x = \text{fold}' \text{ls}') \implies$
 $(x = [] = (\text{ls}' = [])) \implies$
 $(\exists zs \text{zs}'.$
 $\text{list-split}' zs x \wedge \text{list-split}' \text{zs}' \text{ls}' \wedge (\forall n m. n < \text{length } x \longrightarrow m <$
 $\text{length } \text{ls}' \longrightarrow zs ! n ! m = \text{zs}' ! m ! n) \wedge \text{list-all } (\lambda l. \text{length } l = \text{length } \text{ls}') zs \wedge$
 $\text{list-all } (\lambda l. \text{length } l = \text{length } x) \text{zs}') \implies$
 $2 < \text{length } xs \implies \exists zs \text{zs}'.$
 $\text{list-split}' zs xs \wedge$
 $\text{list-split}' \text{zs}' ys \wedge$
 $(\forall n m. n < \text{length } xs \longrightarrow m < \text{length } ys \longrightarrow zs ! n ! m = \text{zs}' ! m ! n) \wedge$
 $\text{list-all } (\lambda l. \text{length } l = \text{length } ys) zs \wedge \text{list-all } (\lambda l. \text{length } l = \text{length } xs) \text{zs}'$

proof –

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

```

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') ∧
  list-split' zs' ys ∧
  (∀ n m. n < length ((x + x') # xs') → m < length ys → zs ! n ! m = zs'
! m ! n) ∧
  list-all (λl. length l = length ys) zs ∧ list-all (λl. length l = length ((x + x')
# xs')) zs'
using ind[of (x + x') # xs' ys] eq-app eq-nil
unfolding xs-def
by auto
have ∃ zsa zs'.
  list-split' zsa [x, x'] ∧
  list-split' zs' (hd zs) ∧
  (∀ n m. n < length [x, x'] → m < length (hd zs) → zsa ! n ! m = zs' ! m !
n) ∧ list-all (λl. length l = length (hd zs)) zsa ∧ list-all (λl. 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'] ≠ [] ∧ hd zs ≠ [] → 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'] ∧ list-split' zs1' (hd zs)
∧
  (∀ n m. n < length [x, x'] → m < length (hd zs) → zs1 ! n ! m = zs1' ! m !
n) ∧ list-all (λl. length l = length (hd zs)) zs1 ∧ list-all (λl. 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 ∧m n. 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
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':  $\exists$  zs1'e1 zs1'e2 zs1'l. zs1' = [zs1'e1, zs1'e2] # zs1'l  $\wedge$ 
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  $\wedge$  zs1'e1 + zs1'e2 = ze
      by auto
    show list-split' (map2 ( $\lambda$ x 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 ( $\lambda$ x y. y ! 0 # y !
Suc 0 # tl x) zss' zs1'l) (y # ys)
    proof (rule list-split'.intros)
      show list-split' (map2 ( $\lambda$ x 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  $\bigwedge$ m. m < length ys  $\longrightarrow$  map tl zs ! 0 ! m = zss' ! m ! 0
        proof -
          fix m show m < length ys  $\longrightarrow$  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
      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 ( $\lambda$ l. length l = length [x, x']) zs1'l
        using len zs1'-def by simp
    next
      show list-all ( $\lambda$ l. 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  $\neq$  []  $\implies$  map tl zs  $\neq$  []
        unfolding zs-def by simp
    qed
  next

```

```

    show  $y = \text{fold}' (zs1'e1 \# zs1'e2 \# \text{tl } z)$ 
      using y-fold z-app zs1'-def by simp
  next
    show  $zs1'e1 \# zs1'e2 \# \text{tl } z \neq []$  by simp
  qed
  show ?thesis
    using goal H
    unfolding zs'-def ze-def trans'
    by simp
  qed
  qed
  have Heq:  $\text{length } zs' = \text{length } zs1'$ 
    using ind1 ind2
    by (cases zs) (auto dest!: list-split'-length)
  have  $\text{length } zs' = \text{length } zs1' \implies \text{map2 } (\lambda x y. y ! 0 \# y ! \text{Suc } 0 \# \text{tl } x) zs' zs1' ! m ! 0 = zs1' ! m ! 0$  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:  $\text{map2 } (\lambda x y. y ! 0 \# y ! \text{Suc } 0 \# \text{tl } x) zs' zs1' ! m ! 0 = zs1' ! m ! 0$  for m
    using Heq
    by blast
  have  $\text{length } zs' = \text{length } zs1' \implies \text{map2 } (\lambda x y. y ! 0 \# y ! \text{Suc } 0 \# \text{tl } x) zs' zs1' ! m ! \text{Suc } 0 = zs1' ! m ! \text{Suc } 0$  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:  $\bigwedge m. \text{map2 } (\lambda x y. y ! 0 \# y ! \text{Suc } 0 \# \text{tl } x) zs' zs1' ! m ! \text{Suc } 0 = zs1' ! m ! \text{Suc } 0$ 
    using Heq
    by blast
  have H3-3:  $n \geq 2 \implies m < \text{length } zs' \implies \text{length } zs' = \text{length } zs1' \implies n \leq \text{length } (zs' ! m) \implies \text{map2 } (\lambda x y. y ! 0 \# y ! \text{Suc } 0 \# \text{tl } x) zs' zs1' ! m ! n = zs' ! m ! (n - 1)$  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:  $n < \text{length } xs \implies m < \text{length } zs \implies \text{list-all } (\lambda l. \text{length } l = \text{Suc } (\text{length } xs)) \text{ } zs \implies \text{Suc } (\text{Suc } n) \leq \text{length } (zs ! m)$ 
  for  $n \ m$  and  $xs :: 'a \ \text{list}$  and  $zs :: 'b \ \text{list}$ 
proof (induction zs arbitrary:  $n \ m \ xs$ )
  case (Cons a zs)
  then show ?case
    by(cases m; simp)
qed simp
have H3:  $\forall n \ m. n < \text{length } xs \longrightarrow m < \text{length } ys \longrightarrow (zs1 ! 0 \# zs1 ! 1 \# \text{tl } zs) ! n ! m = \text{map2 } (\lambda zs' \ y. y ! 0 \# y ! 1 \# \text{tl } zs') \ zs' \ zs1' ! m ! n$ 
proof (intro allI impI)
  fix  $n \ m$ 
  assume  $hn: n < \text{length } xs$  and  $hm: m < \text{length } ys$ 
  show  $(zs1 ! 0 \# zs1 ! 1 \# \text{tl } zs) ! n ! m = \text{map2 } (\lambda zs' \ y. y ! 0 \# y ! 1 \# \text{tl } zs') \ zs' \ zs1' ! m ! n$ 
  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  $\langle n = 0 \rangle$  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  $\langle n = \text{Suc } n' \rangle \langle n' = 0 \rangle$  by (simp add: H3-2[symmetric])
  next
  case (Suc n'')
  have  $hn2: \text{Suc } (\text{Suc } n'') \geq 2$  by simp
  have  $hm2: m < \text{length } zs'$  using ind1 hm by (auto dest: list-split'-length)
  have  $hlen: \text{length } zs' = \text{length } zs1'$  using Heq by simp
  have  $hlen2: \text{Suc } (\text{Suc } n'') \leq \text{length } (zs' ! m)$ 
    using ind1 xs-def  $\langle n = \text{Suc } n' \rangle \langle n' = \text{Suc } n'' \rangle$  hn hm2
    by(fastforce intro: H3-help dest!: list-split'-length)
  have  $\text{step1: } \text{map2 } (\lambda zs' \ y. y ! 0 \# y ! 1 \# \text{tl } zs') \ zs' \ zs1' ! m ! n = zs' ! m ! (n - 1)$ 
    using H3-3[OF hn2 hm2 hlen hlen2]  $\langle n = \text{Suc } n' \rangle \langle n' = \text{Suc } n'' \rangle$  by
    simp
  have  $\text{ntl: } \text{tl } zs ! n'' = zs ! \text{Suc } n''$ 
    using ind1 xs-def  $\langle n = \text{Suc } n' \rangle \langle n' = \text{Suc } n'' \rangle$  hn
    by(fastforce intro: nth-tl dest: list-split'-length)
  have  $\text{step2: } (zs1 ! 0 \# zs1 ! 1 \# \text{tl } zs) ! n ! m = zs' ! m ! (n - 1)$ 
    using ind1 hm ntl xs-def  $\langle n = \text{Suc } n' \rangle \langle n' = \text{Suc } n'' \rangle$  hn

```

```

      by(fastforce dest: list-split'-length)
      with step1 show ?thesis by simp
    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:  $\forall n m. n < \text{length } xs \longrightarrow m < \text{length } ys \longrightarrow ?zs' ! n ! m = ?zs' !$ 
  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 ( $\lambda l. \text{length } l = \text{length } ys$ ) ?zs
    using ref unfolding ys-def2 ys'-def
    by (cases e11; cases e12; cases e21; cases e22) simp-all
  have len-zs': list-all ( $\lambda l. \text{length } l = \text{length } xs$ ) ?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:  $2 < \text{length } xs$ 
show ?case
  using gt3-cases len ind eq-app eq-nil
  by presburger
next
case gt3'
assume len:  $2 < \text{length } ys$ 
have (ys = []) = (xs = [])
  using eq-nil by simp
moreover have  $ys \neq [] \wedge xs \neq [] \longrightarrow \text{fold}' ys = \text{fold}' xs$ 
  using eq-app by simp
moreover have  $\text{length } x + \text{length } ls' < \text{length } ys + \text{length } xs \implies$ 
 $x \neq [] \wedge ls' \neq [] \longrightarrow \text{fold}' x = \text{fold}' ls' \implies$ 
 $(x = []) = (ls' = []) \implies$ 
 $\exists zs \ zs'$ .
  list-split' zs x  $\wedge$ 
  list-split' zs' ls'  $\wedge$ 
  ( $\forall n m. n < \text{length } x \longrightarrow m < \text{length } ls' \longrightarrow zs ! n ! m = zs' ! m ! n$ )  $\wedge$ 
  list-all ( $\lambda l. \text{length } l = \text{length } ls'$ ) zs  $\wedge$  list-all ( $\lambda l. \text{length } l = \text{length } x$ ) 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 $\exists zs\ zs'. \text{list-split } zs\ xs \wedge \text{list-split } zs'\ ys \wedge \text{mset } zs = \text{mset } zs'$
proof –
have *foldl-eq*: $\text{Some } x + \text{foldl } (+) (\text{Some } b) (\text{map } (\lambda x. \text{Some } (\text{snd } x))\ zs) = \text{foldl } (+) (\text{Some } x + \text{Some } b) (\text{map } (\lambda x. \text{Some } (\text{snd } x))\ zs)$ **for** *x b* **and** $zs :: ('a \times 'w :: \text{ab-semigroup-add})\ \text{list}$
by (*induction zs arbitrary: x b*) (*auto simp add: plus-option-def add.left-commute add.assoc*)
have *filter-eq*: $\text{filter } (\lambda(x, -). a = x)\ xs = [] = (\text{weight } (\text{abs-wset } xs)\ a = \text{None})$
for $xs :: ('a \times 'w :: \text{ref-ab-semigroup-add})\ \text{list}$ **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*: $\text{filter } (\lambda(x, -). a = x)\ xs \neq [] \implies \text{weight } (\text{abs-wset } xs)\ a = \text{fold}' (\text{map } (\lambda x. \text{Some } (\text{snd } x))\ (\text{filter } (\lambda(x, -). a = x)\ xs))$
for $xs :: ('a \times 'w :: \text{ref-ab-semigroup-add})\ \text{list}$ **and** *a*
proof –
assume *hne*: $\text{filter } (\lambda(x, -). a = x)\ xs \neq []$
obtain *b list* **where** *hfl*: $\text{filter } (\lambda(x, -). a = x)\ xs = b \# \text{list}$
using *hne* **by** (*cases filter } (\lambda(x, -). a = x)\ xs*) *auto*
have *step*: $\text{weight } (\text{abs-wset } xs)\ a = \text{fold}' (\text{map } (\lambda x. \text{Some } (\text{snd } x))\ (\text{filter } (\lambda(x, -). a = x)\ xs))$
proof –
obtain *b1 b2* **where** *b-def*: $b = (b1, b2)$ **by** *fastforce*
have *rhs-eq*: $\text{fold}' (\text{map } (\lambda x. \text{Some } (\text{snd } x))\ (\text{filter } (\lambda(x, -). a = x)\ xs)) = \text{foldl } (+) (\text{Some } b2) (\text{map } (\lambda x. \text{Some } (\text{snd } x))\ \text{list})$
unfolding *hfl b-def* **by** *simp*
have *lhs-eq*: $\text{weight } (\text{abs-wset } xs)\ a = \text{fold } (\lambda(-, w)\ y. \text{Some } w + y) (\text{filter } (\lambda(x, -). a = x)\ xs)\ \text{None}$
by (*simp add: weight.abs-eq*)
have *conn*: $\text{fold } (\lambda(-, w)\ y. \text{Some } w + y) ((b1, b2) \# \text{list})\ \text{None} = \text{foldl } (+) (\text{Some } b2) (\text{map } (\lambda x. \text{Some } (\text{snd } x))\ \text{list})$
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*: $\forall n < \text{length } zs. \forall m < \text{length } zs'. zs ! n ! m = zs' ! m ! n \implies \text{list-all } (\lambda l. \text{length } l = \text{length } zs')\ zs \implies \text{list-all } (\lambda l. \text{length } l = \text{length } zs)\ zs'$

```

⇒ 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:  $\bigwedge zs'. \forall n < \text{length } zs. \forall m < \text{length } zs'. (zs ! n ! m :: ('w option))$ 
= zs' ! m ! n ⇒ list-all ( $\lambda l. \text{length } l = \text{length } zs'$ ) zs ⇒ list-all ( $\lambda l. \text{length } l =$ 
length zs) zs' ⇒ mset (concat zs) = mset (concat zs')
  and trans:  $\forall n < \text{length } (z \# zs). \forall m < \text{length } zs'. (z \# zs) ! n ! m = (zs' ! m$ 
! n :: ('w option))
  and len1: list-all ( $\lambda l. \text{length } (l :: ('w option) \text{ list}) = \text{length } (zs' :: ('w option)$ 
list list)) (z # zs)
  and len2: list-all ( $\lambda l. \text{length } (l :: ('w option) \text{ list}) = \text{length } ((z :: ('w option)$ 
list) # zs)) zs'
  have trans':  $\bigwedge n m. n < \text{length } (z \# zs) \Rightarrow m < \text{length } zs' \Rightarrow (z \# zs) ! n !$ 
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:  $\forall l \in \text{set } zs'. \text{hd } l = l ! 0$ 
    proof
      fix l assume l ∈ set zs'
      then have length l = Suc (length zs) using len2 by (simp add: list-all-iff)
      then have l ≠ [] 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  $\forall n < \text{length } zs. \forall m < \text{length } (\text{map } \text{tl } zs'). zs ! n ! m = \text{map } \text{tl } zs' ! m ! n$ 
    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 $\exists zs\ zs'. \text{list-split}'\ zs\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ xs)) \wedge$
 $\text{list-split}'\ zs'\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ ys)) \wedge$
 $(\forall n\ m.\ n < \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ xs)) \longrightarrow m$
 $< \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ ys)) \longrightarrow zs\ !\ n\ !\ m = zs'\ !\ m$
 $!\ n) \wedge$
 $\text{list-all}\ (\lambda l.\ \text{length}\ l = \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)$
 $ys)))\ zs \wedge$
 $\text{list-all}\ (\lambda l.\ \text{length}\ l = \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)$
 $xs)))\ zs'$

proof (*intro list-split'-exist*[**where** $xs = (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).$
 $a = x)\ xs)$ **and** $ys = (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ ys))$])

show $(\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ xs) \neq [] \wedge \text{map}\ (\text{Some}\ \circ$
 $\text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ ys) \neq []) \longrightarrow$
 $\text{fold}'\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ xs)) = \text{fold}'\ (\text{map}\ (\text{Some}$
 $\circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ ys))$

proof

assume $H: \text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ xs) \neq [] \wedge$
 $\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ ys) \neq []$

from *wset-eq* **have** $eq': \text{sum-key}\ xs\ a = \text{sum-key}\ ys\ a$

unfolding *eq-wset-def* **by** *auto*

from H **obtain** $fx\ fxs\ fy\ fys$

where $fx\text{-def}: \text{filter}\ (\lambda(x, -).\ a = x)\ xs = fx\ \# \ fxs$
and $fy\text{-def}: \text{filter}\ (\lambda(x, -).\ a = x)\ ys = fy\ \# \ fys$
by (*auto simp: neq-Nil-conv*)

obtain $ax\ wx$ **where** $fx\text{-prod}: fx = (ax, wx)$ **by** (*cases fx*)

obtain $ay\ wy$ **where** $fy\text{-prod}: fy = (ay, wy)$ **by** (*cases fy*)

show $\text{fold}'\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ xs)) =$
 $\text{fold}'\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ ys))$
using eq' **unfolding** $fx\text{-def}\ fy\text{-def}\ fx\text{-prod}\ fy\text{-prod}$
by (*simp add: foldl-conv-fold fold-eq*)

qed

next

show $(\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ xs) = []) =$
 $(\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ ys) = [])$

proof –

have $key: \text{sum-key}\ xs\ a = \text{sum-key}\ ys\ a$

using *wset-eq* **unfolding** *eq-wset-def* **by** *simp*

have $none\text{-iff}: (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x, -).\ a = x)\ zs) = []) =$
 $(\text{sum-key}\ zs\ a = \text{None})$

for $zs :: ('a \times 'w :: \text{ref-ab-semigroup-add})\ \text{list}$ **proof** (*induction*
 zs)

case *Nil*

then show *?case* **by** *simp*

next

case (*Cons h t*)

obtain $k\ v$ **where** $h\text{-def}: h = (k, v)$
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
  then have  $\forall a. \exists zs\ zs'. \text{list-split}'\ zs\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ xs)) \wedge \text{list-split}'\ zs'\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ ys)) \wedge \text{mset}\ (\text{concat}\ zs) = \text{mset}\ (\text{concat}\ zs')$ 
  proof -
    show  $\forall a. \exists zs\ zs'. \text{list-split}'\ zs\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ xs)) \wedge \text{list-split}'\ zs'\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ ys)) \wedge \text{mset}\ (\text{concat}\ zs) = \text{mset}\ (\text{concat}\ zs')$ 
    proof (rule allI)
      fix a
      obtain zs-a zs'-a where
        S1:  $\text{list-split}'\ zs\text{-}a\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ xs))$ 
        and S2:  $\text{list-split}'\ zs'\text{-}a\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ ys))$ 
        and Strans:  $\forall n\ m. n < \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ xs)) \longrightarrow$ 
 $m < \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ ys)) \longrightarrow$ 
 $zs\text{-}a\ !\ n\ !\ m = zs'\text{-}a\ !\ m\ !\ n$ 
        and Slen1:  $\text{list-all}\ (\lambda l. \text{length}\ l = \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ ys)))\ zs\text{-}a$ 
        and Slen2:  $\text{list-all}\ (\lambda l. \text{length}\ l = \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ xs)))\ zs'\text{-}a$ 
        using Hprev by blast
        have Llen1:  $\text{length}\ zs\text{-}a = \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ xs))$ 
        by (rule list-split'-length[OF S1])
        have Llen2:  $\text{length}\ zs'\text{-}a = \text{length}\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ ys))$ 
        by (rule list-split'-length[OF S2])
        have S3:  $\text{mset}\ (\text{concat}\ zs\text{-}a) = \text{mset}\ (\text{concat}\ zs'\text{-}a)$ 
        using Strans Slen1 Slen2 Llen1 Llen2 ind-help by metis
        show  $\exists zs\ zs'. \text{list-split}'\ zs\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ xs)) \wedge \text{list-split}'\ zs'\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ ys)) \wedge \text{mset}\ (\text{concat}\ zs) = \text{mset}\ (\text{concat}\ zs')$ 
        using S1 S2 S3 by blast
      qed
    qed
  then obtain zs zs' where L1:  $\bigwedge a. \text{list-split}'\ (zs\ a)\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ xs))$  and
    L2:  $\bigwedge a. \text{list-split}'\ (zs'\ a)\ (\text{map}\ (\text{Some}\ \circ\ \text{snd})\ (\text{filter}\ (\lambda(x,-).\ a = x)\ ys))$  and
    L3:  $\bigwedge a. \text{mset}\ (\text{concat}\ (zs\ a)) = \text{mset}\ (\text{concat}\ (zs'\ a))$ 
  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  $H$ :  $\text{size } M = \text{Suc } n$ 
shows  $\exists x w N. M = \text{wupdate } N x (\text{Some } w) \wedge \text{weight } N x = \text{None}$ 
proof –
  have  $H1$ :  $\exists x w. \text{weight } M x = \text{Some } w$ 
  proof –
    obtain  $l$  where  $M\text{-eq}$ :  $M = \text{abs-wset } l$  using  $\text{get-abs-wset}$  by  $\text{blast}$ 
    obtain  $m1\ m2\ l'$  where  $l\text{-eq}$ :  $l = (m1, m2) \# l'$ 
    proof ( $\text{cases } l$ )
      case  $\text{Nil}$ 
        then have  $\text{wset } M = \{\}$  by ( $\text{simp add: } M\text{-eq wset.abs-eq}$ )
        then have  $\text{size } M = 0$  by ( $\text{simp add: size-wset-overloaded-def}$ )
        with  $H$  show  $?thesis$  by  $\text{simp}$ 
      next
        case ( $\text{Cons } m\ l'$ )
          obtain  $m1\ m2$  where  $m = (m1, m2)$  by ( $\text{cases } m$ )
          then show  $?thesis$  using  $\text{that Cons}$  by  $\text{blast}$ 
        qed
        have  $\text{not-none}$ :  $\text{fold } (\lambda(a, w). (+) (\text{Some } w)) (\text{filter } (\lambda(e', -). m1 = e')\ l')$ 
          ( $\text{Some } m2$ )  $\neq \text{None}$ 
          by ( $\text{rule fold-Some}$ )
        obtain  $v$  where  $\text{fold } (\lambda(a, w). (+) (\text{Some } w)) (\text{filter } (\lambda(e', -). m1 = e')\ l')$ 
          ( $\text{Some } m2$ )  $= \text{Some } v$ 
          using  $\text{not-none}$  by ( $\text{cases fold } (\lambda(a, w). (+) (\text{Some } w)) (\text{filter } (\lambda(e', -). m1 = e')\ l')$ 
            ( $\text{Some } m2$ ))  $\text{auto}$ 
          then show  $?thesis$ 
          by ( $\text{auto simp add: } M\text{-eq } l\text{-eq weight.abs-eq}$ )
        qed
      then obtain  $x\ w$  where  $Hxw$ :  $\text{weight } M x = \text{Some } w$  by  $\text{blast}$ 
      show  $?thesis$ 
      proof ( $\text{intro exI conjI}$ )
        show  $M = \text{wupdate } (\text{wupdate } M x \text{None}) x (\text{Some } w)$ 
          by ( $\text{simp add: weight-eq-iff[symmetric] } Hxw$ )
        show  $\text{weight } (\text{wupdate } M x \text{None}) x = \text{None}$ 
          by  $\text{simp}$ 
      qed
    qed

```

```

lemma  $\text{size-wupdate}$ :
  assumes  $\text{size}$ :  $\text{Suc } k = \text{size } (\text{wupdate } N x (\text{Some } w))$ 
  and  $\text{weight}$ :  $\text{weight } N x = \text{None}$ 
  shows  $k = \text{size } N$ 
proof –
  obtain  $l$  where  $N\text{-def}$ :  $N = \text{abs-wset } l$ 
  by ( $\text{metis get-abs-wset}$ )
  have  $\text{list-all } (\lambda(a, -). x \neq a)\ l$ 
  using  $\text{weight}$ 
  unfolding  $N\text{-def weight.abs-eq}$ 
  proof ( $\text{induction } l$ )
    case  $\text{Nil}$  then show  $?case$  by  $\text{simp}$ 

```

```

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:  $\bigwedge M. M = wempty \implies P M$ 
  and add:  $\bigwedge x w M. P M \implies weight M x = None \implies P (wupdate M x (Some w))$ 
  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:  $\bigwedge M. k = size M \implies P M$ 
  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 (λx. 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 (λx. 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 λf. ∃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*: $\text{finite } \{x. \exists y. \text{weight } M \ x = \text{Some } y\}$
 using *type-definition-wset*
 unfolding *type-definition-def* by *auto*

lemma *wadd-assoc*: $\text{wadd } x \ (\text{wadd } y \ z) = \text{wadd } (\text{wadd } x \ y) \ z$
 by *transfer auto*

lemma *wadd-commute*: $\text{wadd } x \ y = \text{wadd } y \ x$
 by *transfer (auto simp: eq-wset-append-sym)*

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

lemma *w-list-all2-split-left-invariance*:
 $\text{list-all2 } (\text{rel-prod } R \ (=)) \ xs \ ys \implies \text{list-split } xs' \ xs \implies$
 $\exists ys'. \text{list-all2 } (\text{rel-prod } R \ (=)) \ xs' \ ys' \wedge \text{list-split } ys' \ ys$
proof (*induction xs arbitrary: ys xs'*)
 fix $ys :: ('c \times 'b) \text{ list}$
 and $xs' :: ('a \times 'b) \text{ list}$
 assume $\text{list-all2 } (\text{rel-prod } R \ (=)) \ [] \ ys$
 and $\text{list-split } xs' \ []$
 then show $\exists ys'. \text{list-all2 } (\text{rel-prod } R \ (=)) \ xs' \ ys' \wedge \text{list-split } ys' \ ys$
 by(*cases xs'; auto elim: list-split.cases*)
next
 fix $x :: 'a \times 'b$
 and $xs :: ('a \times 'b) \text{ list}$
 and $ys :: ('c \times 'b) \text{ list}$
 and $xs' :: ('a \times 'b) \text{ list}$
 assume *ind*: $\bigwedge ys \ (xs' :: ('a \times 'b) \text{ list}). \text{list-all2 } (\text{rel-prod } R \ (=)) \ xs \ ys \implies \text{list-split } xs' \ xs \implies \exists ys'. \text{list-all2 } (\text{rel-prod } R \ (=)) \ xs' \ ys' \wedge \text{list-split } ys' \ ys$
 and *list2-all'*: $\text{list-all2 } (\text{rel-prod } R \ (=)) \ (x \# xs) \ ys$
 and *list-split'*: $\text{list-split } xs' \ (x \# xs)$
 have $ys \neq []$
 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.prem1 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  $\implies$ 
  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:  $\bigwedge$ ys. mset xs = mset ys  $\implies$  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'*: $fold (\lambda(a, w). (+) (Some w)) (x \# xs) w \neq None$
by (*induction xs arbitrary: x w*) (*auto simp: plus-option-case*)

lemma *eq-wset-mset*:

$mset (xs :: ('a \times 'w :: ref-ab-semigroup-add) list) = mset ys \implies eq-wset xs ys$

proof (*induction xs arbitrary: ys*)

fix $ys :: ('a \times 'w) list$

assume $mset [] = mset ys$

then show $eq-wset [] ys$

unfolding *eq-wset-def*

by force

next

fix $x :: 'a \times 'w$

and $xs :: ('a \times 'w) list$

and $ys :: ('a \times 'w) list$

obtain $x' w$ **where** $x-def: x = (x', w)$

by force

assume $ind: \bigwedge ys. mset xs = mset ys \implies eq-wset xs ys$

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 $H1: (\exists x \in set xs. a = fst x) = (fold (\lambda(-, w). (+) (Some w)) (filter (\lambda(a', -). a = a') xs) None \neq None)$ **for** $a :: 'a$ **and** $xs :: ('a \times 'w :: ref-ab-semigroup-add) list$

proof (*cases filter (\lambda(a', -). a = a') xs*)

case Nil

then have $no-match: \forall x \in set xs. a \neq fst x$

```

    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:  $\bigcup (Basic-BNFs.fsts \text{ ' set } xs) = (set \ o \ (map \ fst)) \ xs$  for  $xs :: ('a \times 'w$ 
  :: ref-ab-semigroup-add) list
    by (induction xs) auto
  show  $(Ss :: 'a \ set \ set) \neq \{\} \implies \bigcap Ss \neq \{\} \implies$ 
     $(\bigcap As \in Ss. \{(x, x'). eq-wset \ x \ x'\} \text{ '' } \{x :: ('a \times 'w) \ list. \bigcup (Basic-BNFs.fsts$ 
   $\text{ ' set } x) \subseteq As\} \subseteq \{(x, x'). eq-wset \ x \ x'\} \text{ '' } \{x. \bigcup (Basic-BNFs.fsts \text{ ' set } x) \subseteq \bigcap$ 
   $Ss\} \text{ for } Ss$ 
    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 ::  $\langle ('a \implies bool) \implies ('a, 'w :: ref-ab-semigroup-add) \ wset \implies$ 
 $('a, 'w :: ref-ab-semigroup-add) \ wset \rangle$  is
   $\langle \lambda f \ l. filter \ (\lambda x. f \ (fst \ x)) \ l \rangle$ 
  unfolding eq-wset-def
proof (safe)
  fix f :: 'a  $\implies$  bool and l1 l2 :: ('a  $\times$  'w) list and q :: 'a
  assume H:  $\forall q. sum-key \ l1 \ q = sum-key \ l2 \ q$ 
  show sum-key (filter ( $\lambda x. f \ (fst \ x)$ ) l1) q = sum-key (filter ( $\lambda x. f \ (fst \ x)$ ) 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 ::  $\langle ('a \implies 'b \ option) \implies ('a, 'w :: ref-ab-semigroup-add)$ 
 $wset \implies ('b, 'w :: ref-ab-semigroup-add) \ wset \rangle$  where

```

$wimage\text{-option } f\ ws = wimage\ ((case\text{-option } (SOME\ x.\ True)\ id)\ o\ f)\ (wfilter\ ((case\text{-option } False\ (\lambda\cdot.\ True))\ o\ f)\ ws)$

lemma *rep-wset-wempty[simp]*: $rep\text{-wset}\ wempty = ([\] :: ('a \times ('b :: ref\text{-ab}\text{-semigroup}\text{-add}))\ list)$

unfolding *wempty-def*
using *Quotient-rep-abs[OF Quotient-wset, of [\] :: ('a \times 'b)\ list]*
by (*cases rep-wset (abs-wset ([\] :: ('a \times 'b)\ list))*) *auto*

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

lemma *wempty-if-None*: $(\bigwedge x.\ weight\ w\ x = None) \implies w = wempty$
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]*: $pcr\text{-wset}\ R1\ R2\ (\lambda x.\ None)\ wempty$
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]*: $rel\text{-fun}\ (pcr\text{-wset}\ (=)\ (=))\ (rel\text{-fun}\ (=)\ (=))$
 $(\lambda ws\ x.\ ws\ x)\ weight$
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*: $x \in fst\ 'set\ kvs \longleftrightarrow (\exists v.\ sum\text{-key}\ kvs\ x = Some\ v)$

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 (=) (=)) (=) (λws. {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 (=) (=))) (λx 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 (=) (=))) (λws 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 (=) (=)))) (λws 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 (λx. ¬ 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 (λx. ¬ g x) [])) x*
by *simp*
next
fix *a :: 'a × 'b*
and *l :: ('a × 'b :: ref-ab-semigroup-add) list*
assume *ind: fold (λa. case a of (a, w) ⇒ (+) (Some w)) (filter (λa. case a of (a', uu-) ⇒ x = a') (map (map-prod f id) l)) None = fold (λa. case a of (a, w) ⇒ (+) (Some w)) (filter (λa. case a of (a', uu-) ⇒ x = a') (map (map-prod f id) (filter g l))) None + fold (λa. case a of (a, w) ⇒ (+) (Some w)) (filter (λa. case a of (a', uu-) ⇒ x = a') (map (map-prod f id) (filter (λx. ¬ g x) l))) None*
then show *fold (λa. case a of (a, w) ⇒ (+) (Some w)) (filter (λa. case a of (a', uu-) ⇒ x = a') (map (map-prod f id) (a # l))) None = fold (λa. case a of (a, w) ⇒ (+) (Some w)) (filter (λa. case a of (a', uu-) ⇒ x = a') (map (map-prod f id) (filter g (a # l)))) None + fold (λa. case a of (a, w) ⇒ (+) (Some w)) (filter (λa. case a of (a', uu-) ⇒ x = a') (map (map-prod f id) (filter (λx. ¬ g x) (a # l)))) None*
by (*auto simp add: fold-Some-out add.commute[symmetric] add.assoc[symmetric]*)
qed

have $H3'$: $(\lambda p. (case\ p\ of\ (a',\ uu-) \Rightarrow a = a') \wedge (case\ map\text{-}prod\ f\ id\ p\ of\ (a',\ uu-) \Rightarrow x = a')) = (\lambda(a',\ -). a = a' \wedge f\ a = x)$ **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 $H3$: $fold\ (\lambda(a,\ w). (+)\ (Some\ w))\ (filter\ (\lambda(a',\ -). x = a')\ (map\ (map\text{-}prod\ f\ id)\ (filter\ (\lambda(a',\ -). a = a')\ l)))\ None = fold\ (\lambda(a,\ w). (+)\ (Some\ w))\ (filter\ (\lambda(a',\ -). a = a' \wedge f\ a = x)\ l)\ None$
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 $eq\text{-}wset\ l\ l' \Longrightarrow sum\text{-}key\ (map\ (map\text{-}prod\ f\ id)\ l')\ x = sum\text{-}key\ (map\ (map\text{-}prod\ f\ id)\ l)\ x$
proof (*induction map fst l arbitrary: l l' rule: length-induct*)
fix $l :: ('a \times 'b)\ list$
and $l' :: ('a \times 'b)\ list$
assume $ind: \forall\ ys. length\ ys < length\ (map\ fst\ l) \longrightarrow (\forall\ (la :: ('a \times 'b)\ list). ys = map\ fst\ la \longrightarrow (\forall\ lb. eq\text{-}wset\ la\ lb \longrightarrow sum\text{-}key\ (map\ (map\text{-}prod\ f\ id)\ lb)\ x = sum\text{-}key\ (map\ (map\text{-}prod\ f\ id)\ la)\ x))$
and $l\text{-}l'\text{-}eq: eq\text{-}wset\ (l :: ('a \times 'b)\ list)\ l'$
have $ind: \bigwedge(l' :: ('a \times 'b)\ list)\ l''. length\ (map\ fst\ l') < length\ (map\ fst\ l) \Longrightarrow eq\text{-}wset\ l'\ l'' \Longrightarrow sum\text{-}key\ (map\ (map\text{-}prod\ f\ id)\ l'')\ x = sum\text{-}key\ (map\ (map\text{-}prod\ f\ id)\ l')\ x$
using *ind*
by *metis*
consider $l = [] \mid l \neq []$
by *auto*
then show $fold\ (\lambda(a,\ w). (+)\ (Some\ (w :: 'b)))\ (filter\ (\lambda(a',\ -). x = a')\ (map\ (map\text{-}prod\ f\ id)\ l'))\ None = fold\ (\lambda(a,\ w). (+)\ (Some\ w))\ (filter\ (\lambda(a',\ -). x = a')\ (map\ (map\text{-}prod\ f\ id)\ l))\ None$
proof (*cases, goal-cases nil app*)
case *nil*
have $l'\text{-}def: l' = []$
using $l\text{-}l'\text{-}eq\ fold\text{-}Some(2)$
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** $l\text{-}def: l = (a1,\ a2)\ \#\ l1$
using *app*

by (*metis neq-Nil-conv old.prod.exhaust*)
have $A1'$: $a \neq a1 \implies (\lambda x. \neg (\text{case } x \text{ of } (a', uu-) \Rightarrow a1 = a') \wedge (\text{case } x \text{ of } (a', uu-) \Rightarrow a = a')) = (\lambda(a', -). a = a')$ **for** a
by (*rule ext*) *force*
have $A1$: *eq-wset* (*filter* $(\lambda x. \neg (\text{case } x \text{ of } (a', uu-) \Rightarrow a1 = a'))$ l) (*filter* $(\lambda x. \neg (\text{case } x \text{ of } (a', uu-) \Rightarrow a1 = a'))$ l')
using *l-l'-eq*
unfolding *eq-wset-def filter-filter*
proof (*safe*)
fix a
assume H : $\forall q. \text{sum-key } l \ q = \text{sum-key } l' \ q$
show *fold* $(\lambda(-, w). (+) (\text{Some } w))$ (*filter* $(\lambda x. \neg (\text{case } x \text{ of } (a', -) \Rightarrow a1 = a') \wedge (\text{case } x \text{ of } (a', -) \Rightarrow a = a'))$ l) *None* =
fold $(\lambda(-, w). (+) (\text{Some } w))$ (*filter* $(\lambda x. \neg (\text{case } x \text{ of } (a', -) \Rightarrow a1 = a') \wedge (\text{case } x \text{ of } (a', -) \Rightarrow a = a'))$ l') *None*
using H [*rule-format, of a*]
by (*cases a = a1; simp add: A1'[of a]*)
qed
have $A2$: *length* (*map fst* (*filter* $(\lambda x. \neg (\text{case } x \text{ of } (a', uu-) \Rightarrow a1 = a'))$ 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 (lambda x. neg (case x of (a', uu-) => a1 = a')) l filter (lambda x. neg (case x of (a', uu-) => a1 = a')) l', OF A2]]*
unfolding $H2$ [*of x f l lambda(a', -). a1 = a'*] $H2$ [*of x f l' lambda(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* $(=)$ $(=)$))

$(\lambda f \ M \ b. \text{Finite-Set.fold } (\lambda x. (+) (M \ x)) \ \text{None} \ \{a. M \ a \neq \text{None} \wedge f \ a = b\})$
wimage

proof –

have H : *Finite-Set.fold* $(\lambda x. (+) (\text{weight } ws \ x)) \ \text{None} \ \{a. (\exists y. \text{weight } ws \ a = \text{Some } y) \wedge f \ a = x\} = \text{weight } (\text{abs-wset } (\text{map } (\text{map-prod } f \ \text{id}) (\text{rep-wset } ws))) \ x$
for $f :: 'a \Rightarrow 'b$ **and** $ws :: ('a, 'w :: \text{ref-ab-semigroup-add}) \ \text{wset}$ **and** $x :: 'b$

proof (*induction ws rule: wset.abs-induct*)

fix $l :: ('a \times 'w :: \text{ref-ab-semigroup-add}) \ \text{list}$

have *Finite-Set.fold* $(\lambda x. (+) (\text{sum-key } l \ x)) \ w$

$\{a. (\exists v. \text{sum-key } l \ a = \text{Some } v) \wedge f \ a = x\} =$

fold $(\lambda(a, w). (+) (\text{Some } w))$ (*filter* $(\lambda(a', -). x = a')$ (*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 (λx. (+) (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 (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
    consider f a1 = x | f a1 ≠ x
    by fast
  then show Finite-Set.fold (λx. (+) (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 (λx. (+) (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 (λx. (+) (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 (λx. (+) (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 (λx. (+) (sum-key (a # l) x)) w ({b. (∃ v. sum-key
(a # l) b = Some v) ∧ f b = x} - {a1}) =
            Finite-Set.fold (λx. (+) (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 (λx. (+) (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 (λx. (+) (sum-key l x)) w ({b.
(∃ v. sum-key l b = Some v) ∧ f b = x} - {a1}) =
        Finite-Set.fold (λx. (+) (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 (λx. (+) (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 (λx. (+) (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-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:  $v = \text{None}$  using that by simp
have filter-empty:  $\text{filter } (\lambda aw. ws \text{ (fst } aw) \wedge (\text{case } aw \text{ of } (a', -) \Rightarrow x = a')) \text{ } xs$ 
= []
by (rule filter-False) (auto simp add: False case-prod-beta)
have lhs-None:  $(\text{case fold } (\lambda(a, w). (+) (Some w)) (\text{filter } (\lambda(a', -). x = a') \text{ } xs)$ 
v of

$$\text{None} \Rightarrow \text{None} \mid \text{Some } b' \Rightarrow \text{if } ws \text{ } x \text{ then } \text{Some } b' \text{ else } \text{None}) =$$

 $\text{None}$ 
proof (cases filter } (\lambda(a', -). x = a') \text{ } xs)
case Nil
then show ?thesis by (simp add: v-None)
next
case (Cons h t)
then have fold  $(\lambda(a, w). (+) (Some w)) (\text{filter } (\lambda(a', -). x = a') \text{ } xs) \text{ } v \neq$ 
 $\text{None}$ 
unfolding v-None using fold-some' by metis
then obtain b' where  $\text{fold } (\lambda(a, w). (+) (Some w)) (\text{filter } (\lambda(a', -). x = a')$ 
 $\text{ } xs) \text{ } v = \text{Some } b'$ 
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]:  $wimage \text{ } f \text{ } wempty = wempty$ 
by (transfer) (simp add: eq-wset-def)

lemma wimage-wadd-wsingle:  $wimage \text{ } f \text{ } (wadd \text{ } (wsingle \text{ } x \text{ } w) \text{ } M) = wadd \text{ } (wsingle$ 
 $(f \text{ } x) \text{ } w) \text{ } (wimage \text{ } f \text{ } M)$ 
by (transfer) (auto simp add: eq-wset-def)

lemma wimage-wsingle[simp]:  $wimage \text{ } f \text{ } (wsingle \text{ } x \text{ } w) = (wsingle \text{ } (f \text{ } x) \text{ } w)$ 
by (transfer) (auto simp add: eq-wset-def)

lemma wimage-wadd[simp]:  $wimage \text{ } f \text{ } (wadd \text{ } xs \text{ } ys) = wadd \text{ } (wimage \text{ } f \text{ } xs) \text{ } (wimage$ 
 $f \text{ } ys)$ 
by (transfer) auto

lemma w-image-update:
 $weight \text{ } M \text{ } x = \text{None} \Longrightarrow wimage \text{ } f \text{ } (wupdate \text{ } M \text{ } x \text{ } (Some \text{ } w)) = wadd \text{ } (wsingle \text{ } (f$ 

```

```

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:  $x \in \text{set-wset } ws = (\text{weight } (ws :: ('a, 'w :: \text{ref-ab-semigroup-add}) \text{wset}) x \neq \text{None})$ 
proof -
  have H1:  $\text{filter } (\lambda(a', -). e = a') xs \neq [] \implies \text{sum-key } (\text{map } (\text{map-prod } \text{Inr } \text{id}) xs) (\text{Inr } e) \neq \text{None}$  for  $e :: 'a$  and  $xs :: ('a \times 'w :: \text{ref-ab-semigroup-add}) \text{list}$ 
  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:  $(\forall zs. \text{eq-wset } (\text{map } (\text{map-prod } \text{Inr } \text{id}) (\text{rep-wset } ws)) zs \longrightarrow (\exists zs \in \text{set } zs. \exists zs \in \text{Basic-BNFs.fsts } zs. x \in \text{Basic-BNFs.setr } zs)) = (\exists y. \text{weight } ws x = \text{Some } y)$ 
  proof (rule iffI)
    assume H:  $\forall zs. \text{eq-wset } (\text{map } (\text{map-prod } \text{Inr } \text{id}) (\text{rep-wset } ws)) zs \longrightarrow (\exists z \in \text{set } zs. \exists a \in \text{Basic-BNFs.fsts } z. x \in \text{Basic-BNFs.setr } a)$ 
    have mem:  $\exists aw \in \text{set } (\text{rep-wset } ws). x = \text{fst } aw$ 
    proof -
      from H[THEN spec[where  $x = \text{map } (\text{map-prod } \text{Inr } \text{id}) (\text{rep-wset } ws)$ ], 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  $(a, w) \in \text{set } (\text{rep-wset } ws)$  and  $x = a$  by auto
  then show  $\exists y. \text{weight } ws x = \text{Some } y$  by (blast dest: rep-wset-set)
next
  assume H:  $\exists y. \text{weight } ws x = \text{Some } y$ 
  show  $\forall zs. \text{eq-wset } (\text{map } (\text{map-prod } \text{Inr } \text{id}) (\text{rep-wset } ws)) zs \longrightarrow (\exists zs \in \text{set } zs. \exists zs \in \text{Basic-BNFs.fsts } zs. x \in \text{Basic-BNFs.setr } zs)$ 
  proof (rule allI, rule impI)
    fix zs ::  $((d + 'a) \times 'w) \text{list}$ 
    assume Heq:  $\text{eq-wset } (\text{map } (\text{map-prod } \text{Inr } \text{id}) (\text{rep-wset } ws)) zs$ 
    from H obtain y where Hweight:  $\text{weight } ws x = \text{Some } y$  by blast
    have sk-ne:  $\text{sum-key } (\text{map } (\text{map-prod } \text{Inr } \text{id}) (\text{rep-wset } ws)) (\text{Inr } x) \neq \text{None}$ 
    proof -
      from Hweight have  $\text{weight } ws x \neq \text{None}$  by simp
      then show ?thesis
    
```

```

      unfolding weight-def
      by (cases filter ( $\lambda(a', -). x = a'$ ) (rep-wset ws)) (simp-all add: H1)
    qed
  have sk-zs: sum-key zs (Inr x)  $\neq$  None
    using Heq sk-ne unfolding eq-wset-def by metis
  show  $\exists zs \in set\ zs. \exists zs \in Basic\text{-BNFs.fsts}\ zs. x \in Basic\text{-BNFs.setr}\ zs$ 
  proof -
    from sk-zs obtain v where sum-key zs (Inr x) = Some v by fastforce
    then have Inr x  $\in$  fst ' set zs by (simp add: in-set-conv-sum-key-Some)
    then obtain p where hp: p  $\in$  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
  then show ?thesis unfolding set-wset-def by simp
  qed

```

```

lemma set-wset-alt-def: set-wset ws = {x. weight ws x  $\neq$  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 = ( $\exists xs\ ys. abs\text{-wset}\ xs = x \wedge list\text{-all2}\ (rel\text{-prod}\ R\ (=))\ xs\ ys$ 
 $\wedge abs\text{-wset}\ ys = y$ )
  unfolding wset.in-rel set-wset-alt-def
  proof safe
    fix z :: ('a  $\times$  'b, 'w) wset
    assume {x. weight z x  $\neq$  None}  $\subseteq$  {(x, y). R x y}
    then have list-all2 (rel-prod R (=)) (map ( $\lambda((a,b),c). (a,c)$ ) (rep-wset z)) (map
    ( $\lambda((a,b),c). (b,c)$ ) (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 ( $\lambda((a,b),c). (a,c)$ ) (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 ( $\lambda((a,b),c). (b,c)$ ) (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  $\exists xs\ ys. abs\text{-wset}\ xs = wimage\ fst\ z \wedge list\text{-all2}\ (rel\text{-prod}\ R\ (=))\ xs\ ys$ 
 $\wedge abs\text{-wset}\ ys = wimage\ snd\ z$ 
      by blast
  next
    fix xs :: ('a  $\times$  'w) list and ys :: ('b  $\times$  'w) list
    assume list-all2 (rel-prod R (=)) xs ys

```

```

then obtain  $zs :: (('a \times 'w) \times ('b \times 'w)) \text{ list}$  where  $zs: \text{map fst } zs = xs \text{ map}$ 
 $\text{snd } zs = ys$ 
  set  $zs \subseteq \{(x, y). \text{rel-prod } R (=) x y\}$ 
  unfolding  $\text{list.in-rel}$  by  $\text{blast}$ 
let  $?zs = \text{abs-wset } (\text{map } (\lambda((a, -), (b, w)). ((a, b), w)) zs)$ 
from  $zs(\beta)$  have  $?zs \in \{x. \{a. \text{weight } x a \neq \text{None}\} \subseteq \{(x, y). R x y\}\}$ 
  by  $(\text{induct } zs) (\text{auto simp: weight.abs-eq})$ 
moreover from  $zs(1, \beta)$  have  $\text{wimage fst } ?zs = \text{abs-wset } xs$ 
  by  $(\text{force simp flip: abs-wset-map simp: wset.abs-eq-iff eq-wset-def}$ 
     $\text{intro!: arg-cong2[where } f = \text{sum-key}])$ 
moreover from  $zs(2)$  have  $\text{wimage snd } ?zs = \text{abs-wset } ys$ 
  by  $(\text{force simp flip: abs-wset-map simp: wset.abs-eq-iff eq-wset-def}$ 
     $\text{intro!: arg-cong2[where } f = \text{sum-key}])$ 
ultimately show  $\exists z. z \in \{M. \{xy. \text{weight } M xy \neq \text{None}\} \subseteq \{(x, y). R x y\}\} \wedge$ 
 $\text{wimage fst } z = \text{abs-wset } xs \wedge \text{wimage snd } z = \text{abs-wset } ys$ 
  by  $\text{blast}$ 
qed

```

```

end

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

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.