

# Formalization of Weighted Sets

Mathias Schack Rabing      Dmitriy Traytel

June 25, 2026

## Abstract

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

<b>1 Algebraic Preliminaries</b>	<b>2</b>
<b>2 The Positive Representation</b>	<b>2</b>
<b>3 Basic Operations</b>	<b>4</b>
<b>4 The Splitting Relation</b>	<b>6</b>
<b>5 The Negative Representation</b>	<b>8</b>
<b>6 BNF Registration</b>	<b>9</b>
<b>7 Further Operations</b>	<b>10</b>
<b>8 Switching Between Representations</b>	<b>10</b>

```

theory Weighted-Set
  imports
    HOL-Library.Multiset
begin

```

## 1 Algebraic Preliminaries

```

instantiation option :: (ab-semigroup-add) comm-monoid-add begin
definition zero-option where zero-option = None
definition plus-option where plus-option a b = (case (a, b) of (Some x, Some y)
  ⇒ Some (x + y) | (Some x, None) ⇒ Some x | (None, Some x) ⇒ Some x | - ⇒
  None)
instance
  ⟨proof⟩
end

```

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

```

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

```

```

lemma plus-option-simps [simp]: a + None = a None + a = a Some c + Some d
  = Some (c + d)
  ⟨proof⟩

```

```

lemma plus-option-case: Some e + f = Some (case f of Some f' ⇒ e + f' | None
  ⇒ e) f + Some e = Some (case f of Some f' ⇒ f' + e | None ⇒ e)
  ⟨proof⟩

```

```

instantiation option :: (ref-ab-semigroup-add) ref-ab-semigroup-add begin
instance
  ⟨proof⟩
end

```

## 2 The Positive Representation

```

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

```

```

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

```

```

declare [[typedef-overloaded]]

```

**quotient-type** ('a, 'w) wset = ('a × 'w :: ref-ab-semigroup-add) list / eq-wset  
⟨proof⟩

**lemma** get-abs-wset: ∃ l. M = abs-wset l  
⟨proof⟩

**lemma** fold-Some: fold (λ(a, w). (+) (Some w)) xs (Some w) ≠ None None ≠  
fold (λ(a, w). (+) (Some w)) xs (Some w)  
⟨proof⟩

**lemma** fold-Some': ∃ w'. fold (λ(a, w). (+) (Some w)) xs (Some w) = Some w'  
⟨proof⟩

**lemma** fold-Some-out: fold (λ(a, w). (+) (Some w)) xs (Some w) = (Some w) +  
fold (λ(a, w). (+) (Some w)) xs None  
⟨proof⟩

**lemma** eq-wset-fst:  
assumes H: eq-wset xs xs'  
shows set (map fst xs') = set (map fst xs)  
⟨proof⟩

**lemma** eq-wset-refl[simp]: eq-wset xs xs  
⟨proof⟩

**lemma** eq-wset-sym: eq-wset xs xs' ⇒ eq-wset xs' xs  
⟨proof⟩

**lemma** eq-wset-trans: eq-wset xs ys ⇒ eq-wset ys zs ⇒ eq-wset xs zs  
⟨proof⟩

**lemma** eq-wset-elem-switch: eq-wset (x # x' # xs) (x' # x # xs)  
⟨proof⟩

**lemma** eq-wset-elem-comb: eq-wset ((x,w) # (x,w') # xs) ((x,w + w') # xs)  
⟨proof⟩

**lemma** fold-elem-back-aux: fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a')  
(xs @ e1 # e2 # x's)) w =  
fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a') (xs @ e2 # e1 #  
x's)) w  
for a :: 'c and e1 e2 :: 'c × ('d :: ab-semigroup-add) and xs x's :: ('c × 'd) list  
and w :: 'd option  
⟨proof⟩

**lemma** fold-elem-back: fold (λ(a, w). (+) (Some w)) (filter (λ(a', -). a = a') (xs  
@ e # x's)) w =

$\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a') (xs\ @\ x's\ @\ [e]))\ w$   
 $\langle \text{proof} \rangle$

**lemma** *eq-wset-elem-back*:  $\text{eq-wset } (xs\ @\ e\ \# \ x's)\ (xs\ @\ x's\ @\ [e])$   
 $\langle \text{proof} \rangle$

**lemma** *fold-elem-back'*:  $\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a') (e\ \# \ x's))\ w =$   
 $\text{fold } (\lambda(a, w). (+) (Some\ w)) (\text{filter } (\lambda(a', -). a = a') (x's\ @\ [e]))\ w$   
 $\langle \text{proof} \rangle$

**lemma** *eq-wset-elem-back'*:  $\text{eq-wset } (e\ \# \ x's)\ (x's\ @\ [e])$   
 $\langle \text{proof} \rangle$

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

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

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

**lemma** *eq-wset-elem-remove*:  $\text{eq-wset } xs\ xs' \implies \text{eq-wset } (e\ \# \ xs)\ (e\ \# \ xs')$   
 $\langle \text{proof} \rangle$

**lemma** *eq-wset-append-sym*:  $\text{eq-wset } (xs\ @\ ys)\ (ys\ @\ xs)$   
 $\langle \text{proof} \rangle$

### 3 Basic Operations

**lift-definition** *wempty* ::  $\langle ('a, 'w :: \text{ref-ab-semigroup-add})\ \text{wset} \rangle$  **is**  
 $\langle [] \rangle$   $\langle \text{proof} \rangle$

**lift-definition** *weight* ::  $\langle ('a, 'w :: \text{ref-ab-semigroup-add})\ \text{wset} \Rightarrow 'a \Rightarrow 'w\ \text{option} \rangle$   
**is**  
 $\langle \lambda kxs\ e.\ \text{sum-key } kxs\ e \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *weight-eq-iff*:  $(\forall\ x.\ \text{weight } M\ x = \text{weight } N\ x) = (M = N)$   
 $\langle \text{proof} \rangle$

**lift-definition** *wsingle* ::  $\langle 'a \Rightarrow 'w \Rightarrow ('a, 'w :: \text{ref-ab-semigroup-add})\ \text{wset} \rangle$  **is**  
 $\langle \lambda a\ w.\ [(a, w)] \rangle$   $\langle \text{proof} \rangle$

**lift-definition**  $wset :: \langle 'a, 'w :: \text{ref-ab-semigroup-add} \rangle wset \Rightarrow 'a \text{ set} \rangle$  **is**  
 $\langle \lambda M. \text{set} (\text{map fst } M) \rangle \langle \text{proof} \rangle$

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

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

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

**lift-definition**  $wupdate :: \langle 'a, 'w :: \text{ref-ab-semigroup-add} \rangle wset \Rightarrow 'a \Rightarrow 'w \text{ option}$   
 $\Rightarrow ('a, 'w) wset \rangle$  **is**  
 $\langle \lambda M x w. \text{case } w \text{ of } \text{Some } w' \Rightarrow (x, w') \# (\text{filter } (\lambda(x', -). x \neq x') M) | \text{None} \Rightarrow$   
 $\text{filter } (\lambda(x', -). x \neq x') M \rangle$   
 $\langle \text{proof} \rangle$

**instantiation**  $wset :: (\text{type}, \text{type}) \text{ size}$   
**begin**

**definition** *size-wset where*  
 $\text{size-wset-overloaded-def: size-wset } M = \text{card} (wset M)$   
**instance**  $\langle \text{proof} \rangle$

**end**

**lemma** *weight-wsingle[simp]* :  $\text{weight} (w\text{single } x w) x' = (\text{if } x = x' \text{ then } \text{Some } w$   
 $\text{else } \text{None})$   
 $\langle \text{proof} \rangle$

**lemma** *sum-key-append-aux*:  
 $\text{sum-key} (l1 @ l2) x = \text{sum-key } l1 x + \text{sum-key } l2 x$   
**for**  $l1 l2 :: ('a \times ('w :: \text{ref-ab-semigroup-add})) \text{ list}$   
 $\langle \text{proof} \rangle$

**lemma** *weight-add[simp]* :  $\text{weight} (wadd M1 M2) x = \text{weight } M1 x + \text{weight } M2 x$   
 $\langle \text{proof} \rangle$

**lemma** *weight-wempty[simp]* :  $\text{weight } w\text{empty} = (\lambda - . \text{None})$   
 $\langle \text{proof} \rangle$

**lemma** *weight-wupdate[simp]*:  $\text{weight } (\text{wupdate } M \ x \ w) = (\text{weight } M)(x := w)$   
 ⟨proof⟩

**lemma** *wupdate-wupdate[simp]*:  $\text{wupdate } (\text{wupdate } M \ x \ w) \ x \ w' = \text{wupdate } M \ x \ w'$   
 ⟨proof⟩

## 4 The Splitting Relation

**abbreviation** *fold' where*

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

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

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

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

*Base'*:  $\text{list-split}' [] \ []$   
 | *Split'*:  $\text{list-split}' \ xss \ ys \Longrightarrow y = \text{fold}' \ xs \Longrightarrow xs \neq [] \Longrightarrow \text{list-split}' (xs \# xss) (y \# ys)$

**lemma** *list-split-cons-eq*:  $\text{list-split } xs1 \ xs2 \Longrightarrow \text{list-split } (x \# xs1) \ (x \# xs2)$   
 ⟨proof⟩

**lemma** *list-split-refl[simp]*:  $\text{list-split } xs \ xs$   
 ⟨proof⟩

**lemma** *list-split-comb*:  $\text{list-split } ((x, w) \# (x, w') \# xs) \ ((x, w + w') \# xs)$   
 ⟨proof⟩

**lemma** *list-split-nil[simp]*:  $\text{list-split } xs \ [] = (xs = []) \ \text{list-split } [] \ xs = (xs = [])$   
 ⟨proof⟩

**lemma** *eq-wset-nil[simp]*:  $\text{eq-wset } xs \ [] = (xs = [])$   
 ⟨proof⟩

**lemma** *list-split-eq-wset*:

**assumes** *A*:  $\text{list-split } xs \ ys$

**shows**  $\text{eq-wset } xs \ ys$

⟨proof⟩

**lemma** *list-split-app*:  $\text{list-split } xs \ (ys @ ys') \Longrightarrow \exists \ xs' \ xs''. \ \text{list-split } xs' \ ys \wedge \text{list-split } xs'' \ ys' \wedge xs = xs' @ xs''$

⟨proof⟩

**lemma** *list-split-trans*:  
**assumes**  $H$ : *list-split*  $xs$   $ys$   
**and**  $H1$ : *list-split*  $ys$   $zs$   
**shows** *list-split*  $xs$   $zs$   
 $\langle$ *proof* $\rangle$

**lemma** *list-split'-length*: *list-split'*  $xs$   $ys \implies \text{length } xs = \text{length } ys$   
 $\langle$ *proof* $\rangle$

**lemma** *foldl-assoc*:  $(\bigwedge a b c. f (f a b) c = f a (f b c)) \implies f z (\text{foldl } f y xs) = \text{foldl } f (f z y) xs$   
 $\langle$ *proof* $\rangle$

**lemma** *list-split'-refl*: *list-split'*  $(\text{map } (\lambda x. [x]) xs) xs$   
 $\langle$ *proof* $\rangle$

**fun** *option-list* ::  $('a \text{ option}) \text{ list} \Rightarrow 'a \text{ list}$  **where**  
*option-list*  $[] = []$  |  
*option-list*  $(None \# l) = \text{option-list } l$  |  
*option-list*  $(Some a \# l) = a \# \text{option-list } l$

**lemma** *option-list-eq-filter*:  $\text{filter } ((\neq) \text{ None}) l1 = \text{filter } ((\neq) \text{ None}) l2 \implies \text{option-list } l1 = \text{option-list } l2$   
 $\langle$ *proof* $\rangle$

**lemma** *fold-option-not-none*:  $\text{Some } a = \text{fold}' l \implies l \neq [] \implies (\text{option-list } l) \neq []$   
 $\langle$ *proof* $\rangle$

**lemma** *fold-option*:  $\text{Some } a = \text{fold}' l \implies l \neq [] \implies a = \text{fold}' (\text{option-list } l)$   
 $\langle$ *proof* $\rangle$

**fun** *create-split* ::  $('a \times 'w) \text{ list} \Rightarrow ('a \Rightarrow (('w \text{ option}) \text{ list}) \text{ list}) \Rightarrow ('a \times 'w) \text{ list}$  **where**  
*create-split*  $[] \text{ als} = []$  |  
*create-split*  $((a,-) \# l) \text{ als} = \text{map } (\lambda x. (a,x)) (\text{option-list } (\text{hd } (\text{als } a))) @$   
 $(\text{create-split } l (\text{als}(a := \text{tl}(\text{als } a))))$

**lemma** *list-split'-exist*:  
 $((xs \neq [] \wedge ys \neq []) \longrightarrow \text{fold}' (xs :: (('w :: \text{ref-ab-semigroup-add}) \text{ option}) \text{ list}) = \text{fold}' ys) \implies$   
 $((xs = []) = (ys = [])) \implies$   
 $(\exists zs zs'. \text{list-split}' zs xs \wedge \text{list-split}' zs' ys \wedge (\forall n m. n < \text{length } xs \longrightarrow m < \text{length } ys \longrightarrow zs ! n ! m = zs' ! m ! n) \wedge$   
 $\text{list-all } (\lambda l. \text{length } l = \text{length } ys) zs \wedge \text{list-all } (\lambda l. \text{length } l = \text{length } xs) zs')$   
 $\langle$ *proof* $\rangle$

**lemma** *create-split-count*:

$length\ (zs\ a) = length\ (filter\ (\lambda(x, -).\ a = x)\ xs) \implies$   
 $count\ (mset\ (concat\ (zs\ a)))\ (Some\ w) = count\ (mset\ (create-split\ xs\ zs))\ (a,$   
 $w)$   
**for**  $zs :: 'a \Rightarrow ('w :: ref-ab-semigroup-add)\ option\ list\ list$   
**and**  $a :: 'a$  **and**  $xs :: ('a \times 'w)\ list$  **and**  $w :: 'w$   
(*proof*)

**lemma** *list-split-exist*:

**assumes**  $wset-eq: eq-wset\ xs\ ys$   
**shows**  $\exists\ zs\ zs'. list-split\ zs\ xs \wedge list-split\ zs'\ ys \wedge mset\ zs = mset\ zs'$   
(*proof*)

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

**assumes**  $H: size\ M = Suc\ n$   
**shows**  $\exists\ x\ w\ N. M = wupdate\ N\ x\ (Some\ w) \wedge weight\ N\ x = None$   
(*proof*)

**lemma** *size-wupdate*:

**assumes**  $size: Suc\ k = size\ (wupdate\ N\ x\ (Some\ w))$   
**and**  $weight: weight\ N\ x = None$   
**shows**  $k = size\ N$   
(*proof*)

**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*)

## 5 The Negative Representation

**lemma** *weight-inj*: *inj weight*

(*proof*)

**lemma** *type-definition-wset*: *type-definition weight (inv weight) {f :: 'a  $\Rightarrow$  ('w :: ref-ab-semigroup-add) option. finite {x. f x  $\neq$  None}}*

(*proof*)

**lemma** *weight-finite*: *finite {x.  $\exists y. weight\ M\ x = Some\ y}$*

(*proof*)

**lemma** *wadd-assoc*: *wadd x (wadd y z) = wadd (wadd x y) z*

(*proof*)

**lemma** *wadd-commute*: *wadd x y = wadd y x*

*<proof>*

**lemma** *wadd-wsingle[simp]*:  $wadd (wsingle\ x\ w)\ ws = wupdate\ ws\ x$  (Some  $w + weight\ ws\ x$ )

*<proof>*

**lemma** *w-list-all2-split-left-invariance*:

$list\ all2\ (rel\ prod\ R\ (=))\ xs\ ys \implies list\ split\ xs'\ xs \implies$   
 $\exists\ ys'. list\ all2\ (rel\ prod\ R\ (=))\ xs'\ ys' \wedge list\ split\ ys'\ ys$   
*<proof>*

**lemma** *w-list-all2-split-right-invariance*:

$list\ all2\ (rel\ prod\ R\ (=))\ xs\ ys \implies list\ split\ ys'\ ys \implies$   
 $\exists\ xs'. list\ all2\ (rel\ prod\ R\ (=))\ xs'\ ys' \wedge list\ split\ xs'\ xs$   
*<proof>*

**lemma** *w-list-all2-reorder-left-invariance*:

$list\ all2\ (rel\ prod\ R\ (=))\ xs\ ys \implies list\ split\ xs'\ xs \implies$   
 $\exists\ ys'. list\ all2\ (rel\ prod\ R\ (=))\ xs'\ ys' \wedge eq\ wset\ ys'\ ys$   
*<proof>*

**lemma** *w-list-all2-reorder-right-invariance*:

$list\ all2\ (rel\ prod\ R\ (=))\ xs\ ys \implies list\ split\ ys'\ ys \implies$   
 $\exists\ xs'. list\ all2\ (rel\ prod\ R\ (=))\ xs'\ ys' \wedge eq\ wset\ xs'\ xs$   
*<proof>*

**lemma** *eq-wset-remove1*:  $ListMem\ x\ xs \implies eq\ wset\ (x\ \#\ (remove1\ x\ xs))\ xs$

*<proof>*

**lemma** *wset-mset-list*:

$mset\ (xs :: ('a \times 'w :: ref\ ab\ semigroup\ add)\ list) = mset\ ys \implies$   
 $abs\ wset\ xs = abs\ wset\ ys$   
*<proof>*

**lemma** *fold-some'*:  $fold\ (\lambda(a, w). (+)\ (Some\ w))\ (x\ \#\ xs)\ w \neq None$

*<proof>*

**lemma** *eq-wset-mset*:

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

**lemma** *eq-wset-set-fst*:

**assumes**  $A$ :  $eq\ wset\ xs\ ys$   
**shows**  $fst\ 'set\ xs = fst\ 'set\ ys$   
*<proof>*

## 6 BNF Registration

**lift-bnf** ( $'a, dead\ 'w :: ref\ ab\ semigroup\ add$ )  $wset$

**for** *map: wimage rel: wrel*  
 ⟨*proof*⟩

## 7 Further Operations

**lift-definition** *wfilter* :: ⟨('a ⇒ bool) ⇒ ('a, 'w :: ref-ab-semigroup-add) wset ⇒ ('a, 'w :: ref-ab-semigroup-add) wset⟩ **is**  
 ⟨λ*f l*. *filter* (λ*x*. *f* (fst *x*)) *l*⟩  
 ⟨*proof*⟩

**definition** *wimage-option* :: ⟨('a ⇒ 'b option) ⇒ ('a, 'w :: ref-ab-semigroup-add) wset ⇒ ('b, 'w :: ref-ab-semigroup-add) wset⟩ **where**  
*wimage-option* *f ws* = *wimage* ((*case-option* (*SOME* *x*. *True*) *id*) o *f*) (*wfilter* ((*case-option* *False* (λ-. *True*)) o *f*) *ws*)

**lemma** *rep-wset-wempty[simp]*: *rep-wset wempty* = ([] :: ('a × ('b :: ref-ab-semigroup-add)) *list*)  
 ⟨*proof*⟩

**lemma** *wadd-wsingle-wempty[simp]*: *wadd* (*wsingle* *x w*) *wempty* = *wsingle* *x w*  
 ⟨*proof*⟩

**lemma** *wempty-if-None*: (λ*x*. *weight* *w x* = *None*) ⇒ *w* = *wempty*  
 ⟨*proof*⟩

## 8 Switching Between Representations

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

**lemma** *wempty-transfer[transfer-rule]*: *pcr-wset* *R1 R2* (λ*x*. *None*) *wempty*  
 ⟨*proof*⟩

**lemma** *weight-transfer[transfer-rule]*: *rel-fun* (*pcr-wset* (=) (=)) (*rel-fun* (=) (=)) (λ*ws x*. *ws* *x*) *weight*  
 ⟨*proof*⟩

**lemma** *in-set-conv-sum-key-Some*: *x* ∈ *fst* ' *set* *kvs* ⇔ (∃ *v*. *sum-key* *kvs* *x* = *Some* *v*)  
 ⟨*proof*⟩

**lemma** *wset-transfer[transfer-rule]*: *rel-fun* (*pcr-wset* (=) (=)) (=) (λ*ws*. {*x*. *ws* *x* ≠ *None*}) *wset*  
 ⟨*proof*⟩

**lemma** *wsingle-transfer[transfer-rule]*: *rel-fun* (=) (*rel-fun* (=) (*pcr-wset* (=) (=))) (λ*x w x'*. if *x* = *x'* then *Some* *w* else *None*) *wsingle*  
 ⟨*proof*⟩

**lemma** *wadd-transfer[transfer-rule]*: *rel-fun (pcr-wset (=) (=)) (rel-fun (pcr-wset (=) (=)) (pcr-wset (=) (=)))* ( $\lambda ws ws' x. ws\ x + ws'\ x$ ) *wadd*  
 ⟨*proof*⟩

**lemma** *wupdate-transfer[transfer-rule]*: *rel-fun (pcr-wset (=) (=)) (rel-fun (=) (rel-fun (=) (pcr-wset (=) (=))))* ( $\lambda ws\ x\ w\ x'. \text{if } x = x' \text{ then } w \text{ else } ws\ x'$ ) *wupdate*  
 ⟨*proof*⟩

**lemma** *fold-eq-wset*: *eq-wset l l'  $\implies$  sum-key (map (map-prod f id) l') x = sum-key (map (map-prod f id) l) x*  
 ⟨*proof*⟩

**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*⟩

**lemma** *wfilter-transfer[transfer-rule]*: *rel-fun (rel-fun (=) (=)) (rel-fun (pcr-wset (=) (=)) (pcr-wset (=) (=)))*  
 ( $\lambda f\ M\ b. \text{case } (M\ b, f\ b) \text{ of } (\text{Some } b', \text{True}) \implies \text{Some } b' \mid - \implies \text{None}$ ) *wfilter*  
 ⟨*proof*⟩

**end**

**lemma** *wimage-empty[simp]*: *wimage f wempty = wempty*  
 ⟨*proof*⟩

**lemma** *wimage-wadd-wsingle*: *wimage f (wadd (wsingle x w) M) = wadd (wsingle (f x) w) (wimage f M)*  
 ⟨*proof*⟩

**lemma** *wimage-wsingle[simp]*: *wimage f (wsingle x w) = (wsingle (f x) w)*  
 ⟨*proof*⟩

**lemma** *wimage-wadd[simp]*: *wimage f (wadd xs ys) = wadd (wimage f xs) (wimage f ys)*  
 ⟨*proof*⟩

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

**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*:  $abs-wset (rep-wset x) = x$   
 $\langle proof \rangle$

**lemma** *abs-wset-cons*:  $abs-wset ((x,w) \# xs) = wadd (wsingle x w) (abs-wset xs)$   
 $\langle proof \rangle$

**lemma** *abs-wset-map*:  $abs-wset (map (map-prod f id) xs) = wimage f (abs-wset xs)$   
 $\langle proof \rangle$

**context begin**

**interpretation** *wset-as-pfun* $\langle proof \rangle$

**lemma** *rep-wset-set*:  
**assumes**  $(a, w) \in set (rep-wset z)$   
**shows**  $\exists y. weight z a = Some y$   
 $\langle proof \rangle$

**lemma** *set-wset-in*:  $x \in set-wset ws = (weight (ws :: ('a, 'w :: ref-ab-semigroup-add) wset) x \neq None)$   
 $\langle proof \rangle$

**lemma** *set-wset-alt-def*:  $set-wset ws = \{x. weight ws x \neq None\}$   
 $\langle proof \rangle$

**lemma** *wrel-alt-def*:  
**fixes**  $x :: ('a, 'w :: ref-ab-semigroup-add) wset$  **and**  $y :: ('b, 'w) wset$   
**shows**  $wrel R x y = (\exists xs ys. abs-wset xs = x \wedge list-all2 (rel-prod R (=)) xs ys \wedge abs-wset ys = y)$   
 $\langle proof \rangle$

**end**

**declare**  $[[typedef-overloaded]]$

**codatatype**  $('a, 'w) wsetinf = WSetInf (('a, 'w) wsetinf + 'a, 'w :: ref-ab-semigroup-add) wset$

**end**

## References

- [1] H. P. Gumm and T. Schröder. Monoid-labeled transition systems. In A. Corradini, M. Lenisa, and U. Montanari, editors, *CMCS 2001*,

volume 44 of *Electronic Notes in Theoretical Computer Science*, pages  
185–204. Elsevier, 2001.