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

Contents

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

1 Algebraic Preliminaries

instantiation *option* :: (*ab-semigroup-add*) *comm-monoid-add* **begin**
definition *zero-option* **where** *zero-option* = *None*
definition *plus-option* **where** *plus-option a b* = (*case* (*a*, *b*) *of* (*Some x*, *Some y*) ⇒ *Some* (*x* + *y*) | (*Some x*, *None*) ⇒ *Some x* | (*None*, *Some x*) ⇒ *Some x* | - ⇒ *None*)
instance
⟨*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* (λ(-,*w*) *y*. *Some w* + *y*) (*filter* (λ(*e'*,-). *e* = *e'*) *kxs*) *None*

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

declare [[*typedef-overloaded*]]

quotient-type $('a, 'w)$ *wset* $= ('a \times 'w :: \textit{ref-ab-semigroup-add})$ *list* / *eq-wset*
⟨*proof*⟩

lemma *get-abs-wset*: $\exists\, l.\ M = \textit{abs-wset}\ l$
⟨*proof*⟩

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

lemma *fold-Some'*: $\exists\, w'.\ \textit{fold}\ (\lambda(a,\ w).\ (+)\ (\textit{Some}\ w))\ xs\ (\textit{Some}\ w) = \textit{Some}\ w'$
⟨*proof*⟩

lemma *fold-Some-out*: $\textit{fold}\ (\lambda(a,\ w).\ (+)\ (\textit{Some}\ w))\ xs\ (\textit{Some}\ w) = (\textit{Some}\ w) + \textit{fold}\ (\lambda(a,\ w).\ (+)\ (\textit{Some}\ w))\ xs\ \textit{None}$
⟨*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*: $\textit{fold}\ (\lambda(a,\ w).\ (+)\ (\textit{Some}\ w))\ (\textit{filter}\ (\lambda(a',\ \text{-}).\ a = a')\ (xs\ @\ e1\ \#\ e2\ \#\ x's))\ w =$
$\textit{fold}\ (\lambda(a,\ w).\ (+)\ (\textit{Some}\ w))\ (\textit{filter}\ (\lambda(a',\ \text{-}).\ a = a')\ (xs\ @\ e2\ \#\ e1\ \#\ x's))\ w$
for $a :: 'c$ **and** *e1 e2* $:: 'c \times ('d :: \textit{ab-semigroup-add})$ **and** *xs x's* $:: ('c \times 'd)\ \textit{list}$ **and** $w :: 'd\ \textit{option}$
⟨*proof*⟩

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

fold (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a′*, -). *a* = *a′*) (*xs* @ *x′s* @ [*e*])) *w*
⟨*proof*⟩

lemma *eq-wset-elem-back*: *eq-wset* (*xs* @ *e* # *x′s*) (*xs* @ *x′s* @ [*e*])
⟨*proof*⟩

lemma *fold-elem-back′*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a′*, -). *a* = *a′*) (*e* # *x′s*)) *w* =
fold (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a′*, -). *a* = *a′*) (*x′s* @ [*e*])) *w*
⟨*proof*⟩

lemma *eq-wset-elem-back′*: *eq-wset* (*e* # *x′s*) (*x′s* @ [*e*])
⟨*proof*⟩

lemma *fold-Some-back*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a′*, -). *a* = *a′*) *xs*) (*Some* (*M* :: - :: *ref-ab-semigroup-add*)) = *sum-key* (*xs* @ [(*a*, *M*)]) *a*
⟨*proof*⟩

lemma *fold-back′*: *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a′*, -). *a* = *a′*) *xs*) (*Some* (*M* :: - :: *ref-ab-semigroup-add*) + *w*) = *fold* (λ(*a*, *w*). (+) (*Some w*)) (*filter* (λ(*a′*, -). *a* = *a′*) (*xs* @ [(*a*,*M*)])) *w*
⟨*proof*⟩

lemma *eq-wset-append*: *eq-wset xs xs′* ⟹ *eq-wset ys ys′* ⟹ *eq-wset* (*xs* @ *ys*) (*xs′* @ *ys′*)
⟨*proof*⟩

lemma *eq-wset-elem-remove*: *eq-wset xs xs′* ⟹ *eq-wset* (*e* # *xs*) (*e* # *xs′*)
⟨*proof*⟩

lemma *eq-wset-append-sym*: *eq-wset* (*xs* @ *ys*) (*ys* @ *xs*)
⟨*proof*⟩

3 Basic Operations

lift-definition *wempty* :: ‹('*a*, '*w* :: *ref-ab-semigroup-add*) *wset*› **is**
‹[]› ⟨*proof*⟩

lift-definition *weight* :: ‹('*a*, '*w* :: *ref-ab-semigroup-add*) *wset* ⇒ '*a* ⇒ '*w option*› **is**
‹λ*kxs e*. *sum-key kxs e*›
⟨*proof*⟩

lemma *weight-eq-iff*: (∀ *x*. *weight M x* = *weight N x*) = (*M* = *N*)
⟨*proof*⟩

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

lift-definition *wset* :: ‹('a, 'w :: *ref-ab-semigroup-add*) *wset* ⇒ 'a *set*› **is**
‹λM. *set* (*map fst M*)› ⟨*proof*⟩

lift-definition *wadd* :: ‹('a, 'w :: *ref-ab-semigroup-add*) *wset* ⇒ ('a, 'w) *wset* ⇒ ('a, 'w) *wset*› **is**
‹λM1 M2. M1 @ M2› ⟨*proof*⟩

lemma *sum-key-wupdate-same*:
sum-key (*case w of None* ⇒ *filter* (λ(x', -). x ≠ x') l
| *Some w'* ⇒ (x, w') # *filter* (λ(x', -). x ≠ x') l) x = w
for l :: ('a × ('w :: *ref-ab-semigroup-add*)) *list*
⟨*proof*⟩

lemma *sum-key-wupdate-diff*:
x ≠ x' ⟹
sum-key (*case w of None* ⇒ *filter* (λ(x'', -). x ≠ x'') l
| *Some w'* ⇒ (x, w') # *filter* (λ(x'', -). x ≠ x'') l) x' = *sum-key* l x'
for l :: ('a × ('w :: *ref-ab-semigroup-add*)) *list*
⟨*proof*⟩

lift-definition *wupdate* :: ‹('a, 'w :: *ref-ab-semigroup-add*) *wset* ⇒ 'a ⇒ 'w *option* ⇒ ('a, 'w) *wset*› **is**
‹λM x w. *case w of Some w'* ⇒ (x,w') # (*filter* (λ(x',-). x ≠ x') M) | *None* ⇒ *filter* (λ(x',-). x ≠ x') M›
⟨*proof*⟩

instantiation *wset* :: (*type*, *type*) *size*
begin

definition *size-wset* **where**
size-wset-overloaded-def: *size-wset M* = *card* (*wset M*)
instance ⟨*proof*⟩

end

lemma *weight-wsingle*[*simp*] : *weight* (*wsingle x w*) x' = (*if x* = x' *then Some w else None*)
⟨*proof*⟩

lemma *sum-key-append-aux*:
sum-key (*l1* @ *l2*) x = *sum-key l1 x* + *sum-key l2 x*
for *l1 l2* :: ('a × ('w :: *ref-ab-semigroup-add*)) *list*
⟨*proof*⟩

lemma *weight-add*[*simp*] : *weight* (*wadd M1 M2*) x = *weight M1 x* + *weight M2 x*
⟨*proof*⟩

lemma *weight-wempty*[*simp*] : *weight wempty* = (λ- . *None*)
⟨*proof*⟩

lemma *weight-wupdate*[*simp*] : *weight* (*wupdate M x w*) = (*weight M*)(*x* := *w*)
⟨*proof*⟩

lemma *wupdate-wupdate*[*simp*]: *wupdate* (*wupdate M x w*) *x w′* = *wupdate M x w′*
⟨*proof*⟩

4 The Splitting Relation

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

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

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

lemma *list-split-cons-eq*: *list-split xs1 xs2* ⟹ *list-split* (*x* # *xs1*) (*x* # *xs2*)
⟨*proof*⟩

lemma *list-split-refl*[*simp*]: *list-split xs xs*
⟨*proof*⟩

lemma *list-split-comb*: *list-split* ((*x*, *w*) # (*x*, *w′*) # *xs*) ((*x*, *w* + *w′*) # *xs*)
⟨*proof*⟩

lemma *list-split-nil*[*simp*]: *list-split xs* [] = (*xs* = []) *list-split* [] *xs* = (*xs* = [])
⟨*proof*⟩

lemma *eq-wset-nil*[*simp*]: *eq-wset xs* [] = (*xs* = [])
⟨*proof*⟩

lemma *list-split-eq-wset*:
assumes *A*: *list-split xs ys*
shows *eq-wset xs ys*
⟨*proof*⟩

lemma *list-split-app*: *list-split xs* (*ys* @ *ys′*) ⟹ ∃ *xs′ xs″*. *list-split xs′ ys* ∧ *list-split xs″ ys′* ∧ *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*
⟨*proof*⟩

lemma *list-split'-length*: *list-split' xs ys* ⟹ *length xs* = *length ys*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *list-split'-refl*: *list-split'* (*map* (λ*x*. [*x*]) *xs*) *xs*
  ⟨*proof*⟩

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*⟩

lemma *fold-option-not-none*: *Some a* = *fold' l* ⟹ *l* ≠ [] ⟹ (*option-list l*) ≠ []
⟨*proof*⟩

lemma *fold-option*: *Some a* = *fold' l* ⟹ *l* ≠ [] ⟹ *a* = *fold'* (*option-list l*)
⟨*proof*⟩

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

lemma *list-split'-exist*:
  ((*xs* ≠ [] ∧ *ys* ≠ []) ⟶ *fold'* (*xs* :: (('*w* :: *ref-ab-semigroup-add*) *option*) *list*) = *fold' ys*) ⟹
    ((*xs* = []) = (*ys* = [])) ⟹
  (∃ *zs zs'*. *list-split' zs xs* ∧ *list-split' zs' ys* ∧ (∀ *n m*. *n* < *length xs* ⟶ *m* < *length ys* ⟶ *zs* ! *n* ! *m* = *zs'* ! *m* ! *n*) ∧
    list-all (λ*l*. *length l* = *length ys*) *zs* ∧ *list-all* (λ*l*. *length l* = *length xs*) *zs'*)
⟨*proof*⟩

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*⟩

lemma *list-split-exist*:
assumes *wset-eq*: *eq-wset xs ys*
shows ∃ *zs zs'*. *list-split zs xs* ∧ *list-split zs' ys* ∧ *mset zs* = *mset zs'*
⟨*proof*⟩

lemma *w-size-eq-Suc-imp-eq-union*:
assumes *H*: *size M* = *Suc n*
shows ∃ *x w N*. *M* = *wupdate N x* (*Some w*) ∧ *weight N x* = *None*
⟨*proof*⟩

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*: ⋀ *M*. *M* = *wempty* ⟹ *P M*
and *add*: ⋀*x w M*. *P M* ⟹ *weight M x* = *None* ⟹ *P* (*wupdate M x* (*Some w*))
shows *P M*
⟨*proof*⟩

5 The Negative Representation

lemma *weight-inj*: *inj weight*
⟨*proof*⟩

lemma *type-definition-wset*: *type-definition weight* (*inv weight*) {*f* :: *'a* ⇒ (*'w* :: *ref-ab-semigroup-add*) *option*. *finite* {*x*. *f x* ≠ *None*}}
⟨*proof*⟩

lemma *weight-finite*: *finite* {*x*. ∃ *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* ⟹ *list-split xs′ xs* ⟹
∃ *ys′*. *list-all2* (*rel-prod R* (=)) *xs′ ys′* ∧ *list-split ys′ ys*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *eq-wset-remove1*: *ListMem x xs* ⟹ *eq-wset* (*x* # (*remove1 x xs*)) *xs*
⟨*proof*⟩

lemma *wset-mset-list*:
mset (*xs* :: (′*a* × ′*w* :: *ref-ab-semigroup-add*) *list*) = *mset ys* ⟹
abs-wset xs = *abs-wset ys*
⟨*proof*⟩

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

lemma *eq-wset-mset*:
mset (*xs* :: (′*a* × ′*w* :: *ref-ab-semigroup-add*) *list*) = *mset ys* ⟹ *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*: (⋀*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* (=) (=))) (λ*ws ws′ x*. *ws x* + *ws′ x*) *wadd*
⟨*proof*⟩

lemma *wupdate-transfer*[*transfer-rule*]: *rel-fun* (*pcr-wset* (=) (=)) (*rel-fun* (=) (*rel-fun* (=) (*pcr-wset* (=) (=)))) (λ*ws x w x′*. *if x* = *x′ then w else ws x′*) *wupdate*
⟨*proof*⟩

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*⟩

lemma *wimage-transfer*[*transfer-rule*]: *rel-fun* (=) (*rel-fun* (*pcr-wset* (=) (=)) (*pcr-wset* (=) (=)))
(λ*f M b*. *Finite-Set.fold* (λ*x*. (+) (*M x*)) *None* {*a*. *M a* ≠ *None* ∧ *f a* = *b*}) *wimage*
⟨*proof*⟩

lemma *wfilter-transfer*[*transfer-rule*]: *rel-fun* (*rel-fun* (=) (=)) (*rel-fun* (*pcr-wset* (=) (=)) (*pcr-wset* (=) (=)))
(λ*f M b*. *case* (*M b*, *f b*) *of* (*Some b′*, *True*) ⇒ *Some b′* | - ⇒ *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*:
weight M x = *None* ⟹ *wimage f* (*wupdate M x* (*Some w*)) = *wadd* (*wsingle* (*f x*) *w*) (*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*
⟨*proof*⟩
lemma *abs-wset-cons*: *abs-wset* ((*x*,*w*) # *xs*) = *wadd* (*wsingle x w*) (*abs-wset xs*)
⟨*proof*⟩

lemma *abs-wset-map*: *abs-wset* (*map* (*map-prod f id*) *xs*) = *wimage f* (*abs-wset xs*)
⟨*proof*⟩

context begin
interpretation *wset-as-pfun*⟨*proof*⟩

lemma *rep-wset-set*:
assumes (*a*, *w*) ∈ *set* (*rep-wset z*)
shows ∃ *y*. *weight z a* = *Some y*
⟨*proof*⟩

lemma *set-wset-in*: *x* ∈ *set-wset ws* = (*weight* (*ws* :: ('*a*, '*w* :: *ref-ab-semigroup-add*) *wset*) *x* ≠ *None*)
⟨*proof*⟩

lemma *set-wset-alt-def*: *set-wset ws* = {*x*. *weight ws x* ≠ *None*}
⟨*proof*⟩

lemma *wrel-alt-def*:
fixes *x* :: ('*a*, '*w* :: *ref-ab-semigroup-add*) *wset* **and** *y* :: ('*b*, '*w*) *wset*
shows *wrel R x y* = (∃ *xs ys*. *abs-wset xs* = *x* ∧ *list-all2* (*rel-prod R* (=)) *xs ys* ∧ *abs-wset ys* = *y*)
⟨*proof*⟩

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.