ZF

Steven Obua

June 8, 2008

theory *Helper*
imports *Main*
begin

lemma *theI2'*: $?!\ x.\ P\ x \Longrightarrow (!!\ x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (THE\ x.\ P\ x)$
⟨*proof*⟩

lemma *in-range-superfluous*: $(z \in range\ f\ \&\ z \in (f\ `\ x)) = (z \in f\ `\ x)$
⟨*proof*⟩

lemma *f-x-in-range-f*: $f\ x \in range\ f$
⟨*proof*⟩

lemma *comp-inj*: $inj\ f \Longrightarrow inj\ g \Longrightarrow inj\ (g\ o\ f)$
⟨*proof*⟩

lemma *comp-image-eq*: $(g\ o\ f)\ `\ x = g\ `\ f\ `\ x$
⟨*proof*⟩

end

theory *HOLZF*
imports *Helper*
begin

typedecl *ZF*

axiomatization
Empty :: *ZF* **and**
Elem :: $ZF \Rightarrow ZF \Rightarrow bool$ **and**
Sum :: $ZF \Rightarrow ZF$ **and**
Power :: $ZF \Rightarrow ZF$ **and**
Repl :: $ZF \Rightarrow (ZF \Rightarrow ZF) \Rightarrow ZF$ **and**
Inf :: *ZF*

constdefs
Upair:: $ZF \Rightarrow ZF \Rightarrow ZF$
$Upair\ a\ b == Repl\ (Power\ (Power\ Empty))\ (\%\ x.\ if\ x = Empty\ then\ a\ else\ b)$
Singleton:: $ZF \Rightarrow ZF$
$Singleton\ x == Upair\ x\ x$
$union :: ZF \Rightarrow ZF \Rightarrow ZF$
$union\ A\ B == Sum\ (Upair\ A\ B)$
SucNat:: $ZF \Rightarrow ZF$
$SucNat\ x == union\ x\ (Singleton\ x)$
$subset :: ZF \Rightarrow ZF \Rightarrow bool$
$subset\ A\ B == !\ x.\ Elem\ x\ A \longrightarrow Elem\ x\ B$

axioms
Empty: $Not\ (Elem\ x\ Empty)$
Ext: $(x = y) = (!\ z.\ Elem\ z\ x = Elem\ z\ y)$
Sum: $Elem\ z\ (Sum\ x) = (?\ y.\ Elem\ z\ y\ \&\ Elem\ y\ x)$
Power: $Elem\ y\ (Power\ x) = (subset\ y\ x)$
Repl: $Elem\ b\ (Repl\ A\ f) = (?\ a.\ Elem\ a\ A\ \&\ b = f\ a)$
Regularity: $A \neq Empty \longrightarrow (?\ x.\ Elem\ x\ A\ \&\ (!\ y.\ Elem\ y\ x \longrightarrow Not\ (Elem\ y\ A)))$
Infinity: $Elem\ Empty\ Inf\ \&\ (!\ x.\ Elem\ x\ Inf \longrightarrow Elem\ (SucNat\ x)\ Inf)$

constdefs
Sep:: $ZF \Rightarrow (ZF \Rightarrow bool) \Rightarrow ZF$
$Sep\ A\ p == (if\ (!x.\ Elem\ x\ A \longrightarrow Not\ (p\ x))\ then\ Empty\ else$
$(let\ z = (\epsilon\ x.\ Elem\ x\ A\ \&\ p\ x)\ in$
$let\ f = \%\ x.\ (if\ p\ x\ then\ x\ else\ z)\ in\ Repl\ A\ f))$

thm *Power*[*unfolded subset-def*]

theorem *Sep*: $Elem\ b\ (Sep\ A\ p) = (Elem\ b\ A\ \&\ p\ b)$
⟨*proof*⟩

lemma *subset-empty*: $subset\ Empty\ A$
⟨*proof*⟩

theorem *Upair*: $Elem\ x\ (Upair\ a\ b) = (x = a \mid x = b)$
⟨*proof*⟩

lemma *Singleton*: $Elem\ x\ (Singleton\ y) = (x = y)$
⟨*proof*⟩

constdefs
Opair:: $ZF \Rightarrow ZF \Rightarrow ZF$
$Opair\ a\ b == Upair\ (Upair\ a\ a)\ (Upair\ a\ b)$

lemma *Upair-singleton*: $(Upair\ a\ a = Upair\ c\ d) = (a = c\ \&\ a = d)$
⟨*proof*⟩

lemma *Upair-fsteq*: $(Upair\ a\ b = Upair\ a\ c) = ((a = b\ \&\ a = c) \mid (b = c))$
⟨*proof*⟩

lemma *Upair-comm*: $Upair\ a\ b = Upair\ b\ a$
⟨*proof*⟩

theorem *Opair*: $(Opair\ a\ b = Opair\ c\ d) = (a = c\ \&\ b = d)$
⟨*proof*⟩

constdefs
$Replacement :: ZF \Rightarrow (ZF \Rightarrow ZF\ option) \Rightarrow ZF$
$Replacement\ A\ f == Repl\ (Sep\ A\ (\%\ a.\ f\ a \neq None))\ (the\ o\ f)$

theorem *Replacement*: $Elem\ y\ (Replacement\ A\ f) = (?\ x.\ Elem\ x\ A\ \&\ f\ x = Some\ y)$
⟨*proof*⟩

constdefs
$Fst :: ZF \Rightarrow ZF$
$Fst\ q == SOME\ x.\ ?\ y.\ q = Opair\ x\ y$
$Snd :: ZF \Rightarrow ZF$
$Snd\ q == SOME\ y.\ ?\ x.\ q = Opair\ x\ y$

theorem *Fst*: $Fst\ (Opair\ x\ y) = x$
⟨*proof*⟩

theorem *Snd*: $Snd\ (Opair\ x\ y) = y$
⟨*proof*⟩

constdefs
$isOpair :: ZF \Rightarrow bool$
$isOpair\ q == ?\ x\ y.\ q = Opair\ x\ y$

lemma *isOpair*: $isOpair\ (Opair\ x\ y) = True$
⟨*proof*⟩

lemma *FstSnd*: $isOpair\ x \Longrightarrow Opair\ (Fst\ x)\ (Snd\ x) = x$
⟨*proof*⟩

constdefs
$CartProd :: ZF \Rightarrow ZF \Rightarrow ZF$
$CartProd\ A\ B == Sum(Repl\ A\ (\%\ a.\ Repl\ B\ (\%\ b.\ Opair\ a\ b)))$

lemma *CartProd*: $Elem\ x\ (CartProd\ A\ B) = (?\ a\ b.\ Elem\ a\ A\ \&\ Elem\ b\ B\ \&\ x = (Opair\ a\ b))$
⟨*proof*⟩

constdefs
$explode :: ZF \Rightarrow ZF\ set$

explode z == { x. Elem x z }

lemma *explode-Empty*: *(explode x = {}) = (x = Empty)*
⟨*proof*⟩

lemma *explode-Elem*: *(x ∈ explode X) = (Elem x X)*
⟨*proof*⟩

lemma *Elem-explode-in*: *⟦ Elem a A; explode A ⊆ B⟧ ⟹ a ∈ B*
⟨*proof*⟩

lemma *explode-CartProd-eq*: *explode (CartProd a b) = (% (x,y). Opair x y) ‘ ((explode a) × (explode b))*
⟨*proof*⟩

lemma *explode-Repl-eq*: *explode (Repl A f) = image f (explode A)*
⟨*proof*⟩

constdefs
Domain :: ZF ⇒ ZF
Domain f == Replacement f (% p. if isOpair p then Some (Fst p) else None)
Range :: ZF ⇒ ZF
Range f == Replacement f (% p. if isOpair p then Some (Snd p) else None)

theorem *Domain*: *Elem x (Domain f) = (? y. Elem (Opair x y) f)*
⟨*proof*⟩

theorem *Range*: *Elem y (Range f) = (? x. Elem (Opair x y) f)*
⟨*proof*⟩

theorem *union*: *Elem x (union A B) = (Elem x A | Elem x B)*
⟨*proof*⟩

constdefs
Field :: ZF ⇒ ZF
Field A == union (Domain A) (Range A)

constdefs
app :: ZF ⇒ ZF => ZF (**infixl** ´ *90*) — function application
f ´ x == (THE y. Elem (Opair x y) f)

constdefs
isFun :: ZF ⇒ bool
isFun f == (! x y1 y2. Elem (Opair x y1) f & Elem (Opair x y2) f ⟶ y1 = y2)

constdefs
Lambda :: ZF ⇒ (ZF ⇒ ZF) ⇒ ZF
Lambda A f == Repl A (% x. Opair x (f x))

lemma *Lambda-app*: $Elem\ x\ A \Longrightarrow (Lambda\ A\ f)'x = f\ x$
⟨*proof*⟩

lemma *isFun-Lambda*: $isFun\ (Lambda\ A\ f)$
⟨*proof*⟩

lemma *domain-Lambda*: $Domain\ (Lambda\ A\ f) = A$
⟨*proof*⟩

lemma *Lambda-ext*: $(Lambda\ s\ f = Lambda\ t\ g) = (s = t\ \&\ (!\ x.\ Elem\ x\ s \longrightarrow f\ x = g\ x))$
⟨*proof*⟩

constdefs
$PFun :: ZF \Rightarrow ZF \Rightarrow ZF$
$PFun\ A\ B == Sep\ (Power\ (CartProd\ A\ B))\ isFun$
$Fun :: ZF \Rightarrow ZF \Rightarrow ZF$
$Fun\ A\ B == Sep\ (PFun\ A\ B)\ (\lambda\ f.\ Domain\ f = A)$

lemma *Fun-Range*: $Elem\ f\ (Fun\ U\ V) \Longrightarrow subset\ (Range\ f)\ V$
⟨*proof*⟩

lemma *Elem-Elem-PFun*: $Elem\ F\ (PFun\ U\ V) \Longrightarrow Elem\ p\ F \Longrightarrow isOpair\ p\ \&\ Elem\ (Fst\ p)\ U\ \&\ Elem\ (Snd\ p)\ V$
⟨*proof*⟩

lemma *Fun-implies-PFun*[*simp*]: $Elem\ f\ (Fun\ U\ V) \Longrightarrow Elem\ f\ (PFun\ U\ V)$
⟨*proof*⟩

lemma *Elem-Elem-Fun*: $Elem\ F\ (Fun\ U\ V) \Longrightarrow Elem\ p\ F \Longrightarrow isOpair\ p\ \&\ Elem\ (Fst\ p)\ U\ \&\ Elem\ (Snd\ p)\ V$
⟨*proof*⟩

lemma *PFun-inj*: $Elem\ F\ (PFun\ U\ V) \Longrightarrow Elem\ x\ F \Longrightarrow Elem\ y\ F \Longrightarrow Fst\ x = Fst\ y \Longrightarrow Snd\ x = Snd\ y$
⟨*proof*⟩

lemma *Fun-total*: $[\![Elem\ F\ (Fun\ U\ V);\ Elem\ a\ U]\!] \Longrightarrow \exists\, x.\ Elem\ (Opair\ a\ x)\ F$
⟨*proof*⟩

lemma *unique-fun-value*: $[\![isFun\ f;\ Elem\ x\ (Domain\ f)]\!] \Longrightarrow ?!\ y.\ Elem\ (Opair\ x\ y)\ f$
⟨*proof*⟩

lemma *fun-value-in-range*: $[\![isFun\ f;\ Elem\ x\ (Domain\ f)]\!] \Longrightarrow Elem\ (f'x)\ (Range\ f)$
⟨*proof*⟩

lemma *fun-range-witness*: $[\![isFun\ f;\ Elem\ y\ (Range\ f)]\!] \Longrightarrow\ ?\ x.\ Elem\ x\ (Domain\ f)\ \&\ f\ ' x = y$
⟨*proof*⟩

lemma *Elem-Fun-Lambda*: $Elem\ F\ (Fun\ U\ V) \Longrightarrow\ ?\ f.\ F = Lambda\ U\ f$
⟨*proof*⟩

lemma *Elem-Lambda-Fun*: $Elem\ (Lambda\ A\ f)\ (Fun\ U\ V) = (A = U\ \&\ (!\ x.\ Elem\ x\ A \longrightarrow Elem\ (f\ x)\ V))$
⟨*proof*⟩

constdefs
is-Elem-of :: $(ZF * ZF)\ set$
is-Elem-of == $\{\ (a,b)\ |\ a\ b.\ Elem\ a\ b\ \}$

lemma *cond-wf-Elem*:
assumes *hyps*:$\forall\ x.\ (\forall\ y.\ Elem\ y\ x \longrightarrow Elem\ y\ U \longrightarrow P\ y) \longrightarrow Elem\ x\ U \longrightarrow P\ x\ Elem\ a\ U$
shows $P\ a$
⟨*proof*⟩
term *P*
term *Sep*
⟨*proof*⟩

lemma *cond2-wf-Elem*:
assumes
special-P: $?\ U.\ !\ x.\ Not(Elem\ x\ U) \longrightarrow (P\ x)$
and *P-induct*: $\forall\ x.\ (\forall\ y.\ Elem\ y\ x \longrightarrow P\ y) \longrightarrow P\ x$
shows
$P\ a$
⟨*proof*⟩

consts
nat2Nat :: $nat \Rightarrow ZF$

primrec
nat2Nat-0[*intro*]: $nat2Nat\ 0 = Empty$
nat2Nat-Suc[*intro*]: $nat2Nat\ (Suc\ n) = SucNat\ (nat2Nat\ n)$

constdefs
Nat2nat :: $ZF \Rightarrow nat$
Nat2nat == $inv\ nat2Nat$

lemma *Elem-nat2Nat-inf*[*intro*]: $Elem\ (nat2Nat\ n)\ Inf$
⟨*proof*⟩

constdefs

Nat :: *ZF*
Nat == *Sep Inf* (λ *N*. *?* *n*. *nat2Nat n* = *N*)

lemma *Elem-nat2Nat-Nat*[*intro*]: *Elem* (*nat2Nat n*) *Nat*
⟨*proof*⟩

lemma *Elem-Empty-Nat*: *Elem Empty Nat*
⟨*proof*⟩

lemma *Elem-SucNat-Nat*: *Elem N Nat* ⟹ *Elem* (*SucNat N*) *Nat*
⟨*proof*⟩

lemma *no-infinite-Elem-down-chain*:
Not (*?* *f*. *isFun f* & *Domain f* = *Nat* & (! *N*. *Elem N Nat* ⟶ *Elem* (*f´*(*SucNat N*)) (*f´N*)))
⟨*proof*⟩

lemma *Upair-nonEmpty*: *Upair a b* ≠ *Empty*
⟨*proof*⟩

lemma *Singleton-nonEmpty*: *Singleton x* ≠ *Empty*
⟨*proof*⟩

lemma *notsym-Elem*: *Not*(*Elem a b* & *Elem b a*)
⟨*proof*⟩

lemma *irreflexiv-Elem*: *Not*(*Elem a a*)
⟨*proof*⟩

lemma *antisym-Elem*: *Elem a b* ⟹ *Not* (*Elem b a*)
⟨*proof*⟩

consts
NatInterval :: *nat* ⇒ *nat* ⇒ *ZF*

primrec
NatInterval n 0 = *Singleton* (*nat2Nat n*)
NatInterval n (*Suc m*) = *union* (*NatInterval n m*) (*Singleton* (*nat2Nat* (*n*+*m*+*1*)))

lemma *n-Elem-NatInterval*[*rule-format*]: ! *q*. *q* <= *m* ⟶ *Elem* (*nat2Nat* (*n*+*q*)) (*NatInterval n m*)
⟨*proof*⟩

lemma *NatInterval-not-Empty*: *NatInterval n m* ≠ *Empty*
⟨*proof*⟩

lemma *increasing-nat2Nat*[*rule-format*]: *0* < *n* ⟶ *Elem* (*nat2Nat* (*n* − *1*)) (*nat2Nat n*)
⟨*proof*⟩

lemma *represent-NatInterval*[*rule-format*]: *Elem x* (*NatInterval n m*) ⟶ (*? u. n* ≤ *u* & *u* ≤ *n*+*m* & *nat2Nat u* = *x*)
⟨*proof*⟩

lemma *inj-nat2Nat*: *inj nat2Nat*
⟨*proof*⟩

lemma *Nat2nat-nat2Nat*[*simp*]: *Nat2nat* (*nat2Nat n*) = *n*
⟨*proof*⟩

lemma *nat2Nat-Nat2nat*[*simp*]: *Elem n Nat* ⟹ *nat2Nat* (*Nat2nat n*) = *n*
⟨*proof*⟩

lemma *Nat2nat-SucNat*: *Elem N Nat* ⟹ *Nat2nat* (*SucNat N*) = *Suc* (*Nat2nat N*)
⟨*proof*⟩

lemma *Elem-Opair-exists*: *? z. Elem x z* & *Elem y z* & *Elem z* (*Opair x y*)
⟨*proof*⟩

lemma *UNIV-is-not-in-ZF*: *UNIV* ≠ *explode R*
⟨*proof*⟩

constdefs
SpecialR :: (*ZF* ∗ *ZF*) *set*
SpecialR ≡ { (*x*, *y*) . *x* ≠ *Empty* ∧ *y* = *Empty*}

lemma *wf SpecialR*
⟨*proof*⟩

constdefs
Ext :: (*'a* ∗ *'b*) *set* ⇒ *'b* ⇒ *'a set*
Ext R y ≡ { *x* . (*x*, *y*) ∈ *R* }

lemma *Ext-Elem*: *Ext is-Elem-of* = *explode*
⟨*proof*⟩

lemma *Ext SpecialR Empty* ≠ *explode z*
⟨*proof*⟩

constdefs
implode :: *ZF set* ⇒ *ZF*
implode == *inv explode*

lemma *inj-explode*: *inj explode*

⟨*proof*⟩

lemma *implode-explode*[*simp*]: *implode* (*explode x*) = *x*
⟨*proof*⟩

constdefs
regular :: (*ZF* ∗ *ZF*) *set* ⇒ *bool*
regular R == ! *A*. *A* ≠ *Empty* ⟶ (*? x*. *Elem x A* & (! *y*. (*y*, *x*) ∈ *R* ⟶ *Not* (*Elem y A*)))
set-like :: (*ZF* ∗ *ZF*) *set* ⇒ *bool*
set-like R == ! *y*. *Ext R y* ∈ *range explode*
wfzf :: (*ZF* ∗ *ZF*) *set* ⇒ *bool*
wfzf R == *regular R* & *set-like R*

lemma *regular-Elem*: *regular is-Elem-of*
⟨*proof*⟩

lemma *set-like-Elem*: *set-like is-Elem-of*
⟨*proof*⟩

lemma *wfzf-is-Elem-of*: *wfzf is-Elem-of*
⟨*proof*⟩

constdefs
SeqSum :: (*nat* ⇒ *ZF*) ⇒ *ZF*
SeqSum f == *Sum* (*Repl Nat* (*f o Nat2nat*))

lemma *SeqSum*: *Elem x* (*SeqSum f*) = (*? n*. *Elem x* (*f n*))
⟨*proof*⟩

constdefs
Ext-ZF :: (*ZF* ∗ *ZF*) *set* ⇒ *ZF* ⇒ *ZF*
Ext-ZF R s == *implode* (*Ext R s*)

lemma *Elem-implode*: *A* ∈ *range explode* ⟹ *Elem x* (*implode A*) = (*x* ∈ *A*)
⟨*proof*⟩

lemma *Elem-Ext-ZF*: *set-like R* ⟹ *Elem x* (*Ext-ZF R s*) = ((*x*,*s*) ∈ *R*)
⟨*proof*⟩

consts
Ext-ZF-n :: (*ZF* ∗ *ZF*) *set* ⇒ *ZF* ⇒ *nat* ⇒ *ZF*

primrec
Ext-ZF-n R s 0 = *Ext-ZF R s*
Ext-ZF-n R s (*Suc n*) = *Sum* (*Repl* (*Ext-ZF-n R s n*) (*Ext-ZF R*))

constdefs
Ext-ZF-hull :: (*ZF* ∗ *ZF*) *set* ⇒ *ZF* ⇒ *ZF*

Ext-ZF-hull R s == SeqSum (Ext-ZF-n R s)

lemma *Elem-Ext-ZF-hull*:
assumes *set-like-R*: *set-like R*
shows *Elem x (Ext-ZF-hull R S) = (? n. Elem x (Ext-ZF-n R S n))*
⟨*proof*⟩

lemma *Elem-Elem-Ext-ZF-hull*:
assumes *set-like-R*: *set-like R*
and *x-hull*: *Elem x (Ext-ZF-hull R S)*
and *y-R-x*: *(y, x) ∈ R*
shows *Elem y (Ext-ZF-hull R S)*
⟨*proof*⟩

lemma *wfzf-minimal*:
assumes *hyps*: *wfzf R C ≠ {}*
shows *∃ x. x ∈ C ∧ (∀ y. (y, x) ∈ R ⟶ y ∉ C)*
⟨*proof*⟩

lemma *wfzf-implies-wf*: *wfzf R ⟹ wf R*
⟨*proof*⟩

lemma *wf-is-Elem-of*: *wf is-Elem-of*
⟨*proof*⟩

lemma *in-Ext-RTrans-implies-Elem-Ext-ZF-hull*:
set-like R ⟹ x ∈ (Ext (R^+) s) ⟹ Elem x (Ext-ZF-hull R s)
⟨*proof*⟩

lemma *implodeable-Ext-trancl*: *set-like R ⟹ set-like (R^+)*
⟨*proof*⟩

lemma *Elem-Ext-ZF-hull-implies-in-Ext-RTrans*[*rule-format*]:
set-like R ⟹ ! x. Elem x (Ext-ZF-n R s n) ⟶ x ∈ (Ext (R^+) s)
⟨*proof*⟩

lemma *set-like R ⟹ Ext-ZF (R^+) s = Ext-ZF-hull R s*
⟨*proof*⟩

lemma *wf-implies-regular*: *wf R ⟹ regular R*
⟨*proof*⟩

lemma *wf-eq-wfzf*: *(wf R ∧ set-like R) = wfzf R*
⟨*proof*⟩

lemma *wfzf-trancl*: *wfzf R ⟹ wfzf (R^+)*
⟨*proof*⟩

lemma *Ext-subset-mono*: *R ⊆ S ⟹ Ext R y ⊆ Ext S y*

⟨*proof*⟩

lemma *set-like-subset*: *set-like R* ⟹ *S* ⊆ *R* ⟹ *set-like S*
⟨*proof*⟩

lemma *wfzf-subset*: *wfzf S* ⟹ *R* ⊆ *S* ⟹ *wfzf R*
⟨*proof*⟩

end

theory *Zet*
imports *HOLZF*
begin

typedef *'a zet* = {*A* :: *'a set* | *A f z*. *inj-on f A* ∧ *f ' A* ⊆ *explode z*}
⟨*proof*⟩

constdefs
zin :: *'a* ⇒ *'a zet* ⇒ *bool*
zin x A == *x* ∈ (*Rep-zet A*)

lemma *zet-ext-eq*: (*A* = *B*) = (! *x*. *zin x A* = *zin x B*)
⟨*proof*⟩

constdefs
zimage :: (*'a* ⇒ *'b*) ⇒ *'a zet* ⇒ *'b zet*
zimage f A == *Abs-zet* (*image f* (*Rep-zet A*))

lemma *zet-def'*: *zet* = {*A* :: *'a set* | *A f z*. *inj-on f A* ∧ *f ' A* = *explode z*}
⟨*proof*⟩

lemma *image-Inv-f-f*: *inj-on f B* ⟹ *A* ⊆ *B* ⟹ (*Inv B f*) *' f ' A* = *A*
⟨*proof*⟩

lemma *image-zet-rep*: *A* ∈ *zet* ⟹ *?* *z* . *g ' A* = *explode z*
⟨*proof*⟩

lemma *Inv-f-f-mem*:
assumes *x* ∈ *A*
shows *Inv A g* (*g x*) ∈ *A*
⟨*proof*⟩

lemma *zet-image-mem*:
assumes *Azet*: *A* ∈ *zet*
shows *g ' A* ∈ *zet*
⟨*proof*⟩

lemma *Rep-zimage-eq*: *Rep-zet* (*zimage f A*) = *image f* (*Rep-zet A*)
⟨*proof*⟩

lemma *zimage-iff*: *zin y* (*zimage f A*) = (*? x. zin x A & y* = *f x*)
⟨*proof*⟩

constdefs
zimplode :: *ZF zet* ⇒ *ZF*
zimplode A == *implode* (*Rep-zet A*)
zexplode :: *ZF* ⇒ *ZF zet*
zexplode z == *Abs-zet* (*explode z*)

lemma *Rep-zet-eq-explode*: *? z. Rep-zet A* = *explode z*
⟨*proof*⟩

lemma *zexplode-zimplode*: *zexplode* (*zimplode A*) = *A*
⟨*proof*⟩

lemma *explode-mem-zet*: *explode z* ∈ *zet*
⟨*proof*⟩

lemma *zimplode-zexplode*: *zimplode* (*zexplode z*) = *z*
⟨*proof*⟩

lemma *zin-zexplode-eq*: *zin x* (*zexplode A*) = *Elem x A*
⟨*proof*⟩

lemma *comp-zimage-eq*: *zimage g* (*zimage f A*) = *zimage* (*g o f*) *A*
⟨*proof*⟩

constdefs
zunion :: *'a zet* ⇒ *'a zet* ⇒ *'a zet*
zunion a b ≡ *Abs-zet* ((*Rep-zet a*) ∪ (*Rep-zet b*))
zsubset :: *'a zet* ⇒ *'a zet* ⇒ *bool*
zsubset a b ≡ *! x. zin x a* ⟶ *zin x b*

lemma *explode-union*: *explode* (*union a b*) = (*explode a*) ∪ (*explode b*)
⟨*proof*⟩

lemma *Rep-zet-zunion*: *Rep-zet* (*zunion a b*) = (*Rep-zet a*) ∪ (*Rep-zet b*)
⟨*proof*⟩

lemma *zunion*: *zin x* (*zunion a b*) = ((*zin x a*) ∨ (*zin x b*))
⟨*proof*⟩

lemma *zimage-zexplode-eq*: *zimage f* (*zexplode z*) = *zexplode* (*Repl z f*)
⟨*proof*⟩

lemma *range-explode-eq-zet*: *range explode* = *zet*

⟨*proof*⟩

lemma *Elem-zimplode*: (*Elem x* (*zimplode z*)) = (*zin x z*)
⟨*proof*⟩

constdefs
zempty :: *'a zet*
zempty ≡ *Abs-zet* {}

lemma *zempty*[*simp*]: ¬ (*zin x zempty*)
⟨*proof*⟩

lemma *zimage-zempty*[*simp*]: *zimage f zempty* = *zempty*
⟨*proof*⟩

lemma *zunion-zempty-left*[*simp*]: *zunion zempty a* = *a*
⟨*proof*⟩

lemma *zunion-zempty-right*[*simp*]: *zunion a zempty* = *a*
⟨*proof*⟩

lemma *zimage-id*[*simp*]: *zimage id A* = *A*
⟨*proof*⟩

lemma *zimage-cong*[*recdef-cong*]: ⟦ *M* = *N*; !! *x*. *zin x N* ⟹ *f x* = *g x* ⟧ ⟹ *zimage f M* = *zimage g N*
⟨*proof*⟩

end

1 Multisets

theory *Multiset*
imports *List*
begin

1.1 The type of multisets

typedef *'a multiset* = {*f*::*'a* => *nat*. *finite* {*x* . *f x* > *0*}}
⟨*proof*⟩

lemmas *multiset-typedef* [*simp*] =
Abs-multiset-inverse Rep-multiset-inverse Rep-multiset
and [*simp*] = *Rep-multiset-inject* [*symmetric*]

definition
Mempty :: *'a multiset* ({#}) **where**
{#} = *Abs-multiset* (λ*a*. *0*)

definition
single :: *'a => 'a multiset* **where**
single a = Abs-multiset (λb. if b = a then 1 else 0)

declare
Mempty-def[*code func del*] *single-def*[*code func del*]

definition
count :: *'a multiset => 'a => nat* **where**
count = Rep-multiset

definition
MCollect :: *'a multiset => ('a => bool) => 'a multiset* **where**
MCollect M P = Abs-multiset (λx. if P x then Rep-multiset M x else 0)

abbreviation
Melem :: *'a => 'a multiset => bool* ((-/ :# -) [*50*, *51*] *50*) **where**
a :# M == 0 < count M a

notation (*xsymbols*)
Melem (**infix** ∈# *50*)

syntax
-MCollect :: *pttrn => 'a multiset => bool => 'a multiset* ((*1*{# - :# -./ -#}))
translations
{#*x* :# *M*. *P*#} == *CONST MCollect M* (λ*x*. *P*)

definition
set-of :: *'a multiset => 'a set* **where**
set-of M = {*x*. *x* :# *M*}

instantiation *multiset* :: (*type*) {*plus*, *minus*, *zero*, *size*}
begin

definition
union-def[*code func del*]:
M + N = Abs-multiset (λ*a*. *Rep-multiset M a + Rep-multiset N a*)

definition
diff-def: *M − N = Abs-multiset* (λ*a*. *Rep-multiset M a − Rep-multiset N a*)

definition
Zero-multiset-def [*simp*]: *0* = {#}

definition
size-def[*code func del*]: *size M = setsum* (*count M*) (*set-of M*)

instance ⟨*proof*⟩

end

definition
multiset-inter :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* (**infixl** #∩ *70*) **where**
multiset-inter A B = *A* − (*A* − *B*)

Multiset Enumeration

syntax
-multiset :: *args* => *'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == {#*x*#} + {#*xs*#}
{#*x*#} == *CONST single x*

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: (λ*a*. *0*) ∈ *multiset*
⟨*proof*⟩

lemma *only1-in-multiset*: (λ*b*. *if b* = *a then 1 else 0*) ∈ *multiset*
⟨*proof*⟩

lemma *union-preserves-multiset*:
M ∈ *multiset* ==> *N* ∈ *multiset* ==> (λ*a*. *M a* + *N a*) ∈ *multiset*
⟨*proof*⟩

lemma *diff-preserves-multiset*:
M ∈ *multiset* ==> (λ*a*. *M a* − *N a*) ∈ *multiset*
⟨*proof*⟩

lemma *MCollect-preserves-multiset*:
M ∈ *multiset* ==> (λ*x*. *if P x then M x else 0*) ∈ *multiset*
⟨*proof*⟩

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset MCollect-preserves-multiset

1.2 Algebraic properties

1.2.1 Union

lemma *union-empty* [*simp*]: *M* + {#} = *M* ∧ {#} + *M* = *M*
⟨*proof*⟩

lemma *union-commute*: *M* + *N* = *N* + (*M*::*'a multiset*)
⟨*proof*⟩

lemma *union-assoc*: (*M* + *N*) + *K* = *M* + (*N* + (*K*::*'a multiset*))
⟨*proof*⟩

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
⟨*proof*⟩

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

instance *multiset* :: (*type*) *comm-monoid-add*
⟨*proof*⟩

1.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
⟨*proof*⟩

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
⟨*proof*⟩

lemma *diff-cancel*: $A - A = \{\#\}$
⟨*proof*⟩

1.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* $\{\#\}$ *a* = *0*
⟨*proof*⟩

lemma *count-single* [*simp*]: *count* $\{\#b\#\}$ *a* = (*if b* = *a then 1 else 0*)
⟨*proof*⟩

lemma *count-union* [*simp*]: *count* (*M* + *N*) *a* = *count M a* + *count N a*
⟨*proof*⟩

lemma *count-diff* [*simp*]: *count* (*M* − *N*) *a* = *count M a* − *count N a*
⟨*proof*⟩

lemma *count-MCollect* [*simp*]:
count $\{\#\ x{:}\#M.\ P\ x\ \#\}$ *a* = (*if P a then count M a else 0*)
⟨*proof*⟩

1.2.4 Set of elements

lemma *set-of-empty* [*simp*]: *set-of* $\{\#\} = \{\}$
⟨*proof*⟩

lemma *set-of-single* [*simp*]: *set-of* $\{\#b\#\} = \{b\}$
⟨*proof*⟩

lemma *set-of-union* [*simp*]: *set-of* (*M* + *N*) = *set-of M* ∪ *set-of N*
⟨*proof*⟩

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of M* = {}) = (*M* = $\{\#\}$)
⟨*proof*⟩

lemma *mem-set-of-iff* [*simp*]: $(x \in \textit{set-of}\ M) = (x :\# M)$
⟨*proof*⟩

lemma *set-of-MCollect* [*simp*]: $\textit{set-of}\ \{\#\ x{:}\#M.\ P\ x\ \#\} = \textit{set-of}\ M \cap \{x.\ P\ x\}$
⟨*proof*⟩

1.2.5 Size

lemma *size-empty* [*simp*,*code func*]: $\textit{size}\ \{\#\} = 0$
⟨*proof*⟩

lemma *size-single* [*simp*,*code func*]: $\textit{size}\ \{\#b\#\} = 1$
⟨*proof*⟩

lemma *finite-set-of* [*iff*]: $\textit{finite}\ (\textit{set-of}\ M)$
⟨*proof*⟩

lemma *setsum-count-Int*:
$\textit{finite}\ A \Longrightarrow \textit{setsum}\ (\textit{count}\ N)\ (A \cap \textit{set-of}\ N) = \textit{setsum}\ (\textit{count}\ N)\ A$
⟨*proof*⟩

lemma *size-union*[*simp*,*code func*]: $\textit{size}\ (M + N{::}'a\ \textit{multiset}) = \textit{size}\ M + \textit{size}\ N$
⟨*proof*⟩

lemma *size-eq-0-iff-empty* [*iff*]: $(\textit{size}\ M = 0) = (M = \{\#\})$
⟨*proof*⟩

lemma *nonempty-has-size*: $(S \neq \{\#\}) = (0 < \textit{size}\ S)$
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: $\textit{size}\ M = \textit{Suc}\ n \Longrightarrow \exists\, a.\ a :\# M$
⟨*proof*⟩

1.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: $(M = N) = (\forall\, a.\ \textit{count}\ M\ a = \textit{count}\ N\ a)$
⟨*proof*⟩

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
⟨*proof*⟩

lemma *single-eq-single* [*simp*]: $(\{\#a\#\} = \{\#b\#\}) = (a = b)$
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: $(M + N = \{\#\}) = (M = \{\#\} \wedge N = \{\#\})$
⟨*proof*⟩

lemma *empty-eq-union* [*iff*]: $(\{\#\} = M + N) = (M = \{\#\} \wedge N = \{\#\})$
⟨*proof*⟩

lemma *union-right-cancel* [*simp*]: $(M + K = N + K) = (M = (N::'a\ multiset))$
⟨*proof*⟩

lemma *union-left-cancel* [*simp*]: $(K + M = K + N) = (M = (N::'a\ multiset))$
⟨*proof*⟩

lemma *union-is-single*:
$(M + N = \{\#a\#\}) = (M = \{\#a\#\} \wedge N=\{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\})$
⟨*proof*⟩

lemma *single-is-union*:
$(\{\#a\#\} = M + N) \longleftrightarrow (\{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N)$
⟨*proof*⟩

lemma *add-eq-conv-diff*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\})$
⟨*proof*⟩

declare *Rep-multiset-inject* [*symmetric*, *simp del*]

instance *multiset* :: (*type*) *cancel-ab-semigroup-add*
⟨*proof*⟩

lemma *insert-DiffM*:
$x \in\# M \Longrightarrow \{\#x\#\} + (M - \{\#x\#\}) = M$
⟨*proof*⟩

lemma *insert-DiffM2*[*simp*]:
$x \in\# M \Longrightarrow M - \{\#x\#\} + \{\#x\#\} = M$
⟨*proof*⟩

lemma *multi-union-self-other-eq*:
$(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
⟨*proof*⟩

lemma *multi-self-add-other-not-self*[*simp*]: $(A = A + \{\#x\#\}) = False$
⟨*proof*⟩

lemma *insert-noteq-member*:
assumes *BC*: $B + \{\#b\#\} = C + \{\#c\#\}$
and *bnotc*: $b \neq c$
shows $c \in\# B$
⟨*proof*⟩

lemma *add-eq-conv-ex*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee (\exists K.\ M = K + \{\#b\#\} \wedge N = K + \{\#a\#\}))$
⟨*proof*⟩

lemma *empty-multiset-count*:
$(\forall x.\ count\ A\ x = 0) = (A = \{\#\})$
⟨*proof*⟩

1.2.7 Intersection

lemma *multiset-inter-count*:
$count\ (A\ \#\cap\ B)\ x = min\ (count\ A\ x)\ (count\ B\ x)$
⟨*proof*⟩

lemma *multiset-inter-commute*: $A\ \#\cap\ B = B\ \#\cap\ A$
⟨*proof*⟩

lemma *multiset-inter-assoc*: $A\ \#\cap\ (B\ \#\cap\ C) = A\ \#\cap\ B\ \#\cap\ C$
⟨*proof*⟩

lemma *multiset-inter-left-commute*: $A\ \#\cap\ (B\ \#\cap\ C) = B\ \#\cap\ (A\ \#\cap\ C)$
⟨*proof*⟩

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *multiset-inter-single*: $a \neq b \Longrightarrow \{\#a\#\}\ \#\cap\ \{\#b\#\} = \{\#\}$
⟨*proof*⟩

lemma *multiset-union-diff-commute*: $B\ \#\cap\ C = \{\#\} \Longrightarrow A + B - C = A - C + B$
⟨*proof*⟩

1.2.8 Comprehension (filter)

lemma *MCollect-empty*[*simp*, *code func*]: $MCollect\ \{\#\}\ P = \{\#\}$
⟨*proof*⟩

lemma *MCollect-single*[*simp*, *code func*]:
$MCollect\ \{\#x\#\}\ P = (if\ P\ x\ then\ \{\#x\#\}\ else\ \{\#\})$
⟨*proof*⟩

lemma *MCollect-union*[*simp*, *code func*]:
$MCollect\ (M+N)\ f = MCollect\ M\ f + MCollect\ N\ f$
⟨*proof*⟩

1.3 Induction and case splits

lemma *setsum-decr*:
finite F ==> (0::nat) < f a ==>
setsum (f (a := f a − 1)) F = (if a∈F then setsum f F − 1 else setsum f F)
⟨*proof*⟩

lemma *rep-multiset-induct-aux*:
assumes *1*: *P (λa. (0::nat))*
and *2*: *!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))*
shows *∀f. f ∈ multiset −−> setsum f {x. f x ≠ 0} = n −−> P f*
⟨*proof*⟩

theorem *rep-multiset-induct*:
f ∈ multiset ==> P (λa. 0) ==>
(!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f
⟨*proof*⟩

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: *P {#}*
and *add*: *!!M x. P M ==> P (M + {#x#})*
shows *P M*
⟨*proof*⟩

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists A\ a.\ M = A + \{\#a\#\}$
⟨*proof*⟩

lemma *multiset-cases* [*cases type*, *case-names empty add*]:
assumes *em*: $M = \{\#\} \Longrightarrow P$
assumes *add*: $\bigwedge N\ x.\ M = N + \{\#x\#\} \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = A + \{\#x\#\}$
⟨*proof*⟩

lemma *multiset-partition*: $M = \{\#\ x{:}\#M.\ P\ x\ \#\} + \{\#\ x{:}\#M.\ \neg\ P\ x\ \#\}$
⟨*proof*⟩

declare *multiset-typedef* [*simp del*]

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
⟨*proof*⟩

1.4 Orderings

1.4.1 Well-foundedness

definition
mult1 :: *('a × 'a) set => ('a multiset × 'a multiset) set* **where**

mult1 r =
{(*N*, *M*). ∃ *a M0 K*. *M* = *M0* + {#*a*#} ∧ *N* = *M0* + *K* ∧
(∀ *b*. *b* :# *K* −−> (*b*, *a*) ∈ *r*)}

definition
mult :: ('*a* × '*a*) *set* => ('*a multiset* × '*a multiset*) *set* **where**
mult r = (*mult1 r*)$^+$

lemma *not-less-empty* [*iff*]: (*M*, {#}) ∉ *mult1 r*
⟨*proof*⟩

lemma *less-add*: (*N*, *M0* + {#*a*#}) ∈ *mult1 r* ==>
(∃ *M*. (*M*, *M0*) ∈ *mult1 r* ∧ *N* = *M* + {#*a*#}) ∨
(∃ *K*. (∀ *b*. *b* :# *K* −−> (*b*, *a*) ∈ *r*) ∧ *N* = *M0* + *K*)
(**is** - ⟹ *?case1* (*mult1 r*) ∨ *?case2*)
⟨*proof*⟩

lemma *all-accessible*: *wf r* ==> ∀ *M*. *M* ∈ *acc* (*mult1 r*)
⟨*proof*⟩

theorem *wf-mult1*: *wf r* ==> *wf* (*mult1 r*)
⟨*proof*⟩

theorem *wf-mult*: *wf r* ==> *wf* (*mult r*)
⟨*proof*⟩

1.4.2 Closure-free presentation

lemma *diff-union-single-conv*: *a* :# *J* ==> *I* + *J* − {#*a*#} = *I* + (*J* − {#*a*#})
⟨*proof*⟩

One direction.

lemma *mult-implies-one-step*:
trans r ==> (*M*, *N*) ∈ *mult r* ==>
∃ *I J K*. *N* = *I* + *J* ∧ *M* = *I* + *K* ∧ *J* ≠ {#} ∧
(∀ *k* ∈ *set-of K*. ∃ *j* ∈ *set-of J*. (*k*, *j*) ∈ *r*)
⟨*proof*⟩

lemma *elem-imp-eq-diff-union*: *a* :# *M* ==> *M* = *M* − {#*a*#} + {#*a*#}
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*: *size M* = *Suc n* ==> ∃ *a N*. *M* = *N* + {#*a*#}
⟨*proof*⟩

lemma *one-step-implies-mult-aux*:
trans r ==>
∀ *I J K*. (*size J* = *n* ∧ *J* ≠ {#} ∧ (∀ *k* ∈ *set-of K*. ∃ *j* ∈ *set-of J*. (*k*, *j*) ∈ *r*))
−−> (*I* + *K*, *I* + *J*) ∈ *mult r*
⟨*proof*⟩

lemma *one-step-implies-mult*:
trans r ==> *J* ≠ {#} ==> ∀ *k* ∈ *set-of K*. ∃ *j* ∈ *set-of J*. (*k*, *j*) ∈ *r*
==> (*I* + *K*, *I* + *J*) ∈ *mult r*
⟨*proof*⟩

1.4.3 Partial-order properties

instantiation *multiset* :: (*order*) *order*
begin

definition
less-multiset-def: *M'* < *M* ⟷ (*M'*, *M*) ∈ *mult* {(*x'*, *x*). *x'* < *x*}

definition
le-multiset-def: *M'* <= *M* ⟷ *M'* = *M* ∨ *M'* < (*M*::'*a multiset*)

lemma *trans-base-order*: *trans* {(*x'*, *x*). *x'* < (*x*::'*a*::*order*)}
⟨*proof*⟩

Irreflexivity.

lemma *mult-irrefl-aux*:
finite A ==> (∀ *x* ∈ *A*. ∃ *y* ∈ *A*. *x* < (*y*::'*a*::*order*)) ⟹ *A* = {}
⟨*proof*⟩

lemma *mult-less-not-refl*: ¬ *M* < (*M*::'*a*::*order multiset*)
⟨*proof*⟩

lemma *mult-less-irrefl* [*elim*!]: *M* < (*M*::'*a*::*order multiset*) ==> *R*
⟨*proof*⟩

Transitivity.

theorem *mult-less-trans*: *K* < *M* ==> *M* < *N* ==> *K* < (*N*::'*a*::*order multiset*)
⟨*proof*⟩

Asymmetry.

theorem *mult-less-not-sym*: *M* < *N* ==> ¬ *N* < (*M*::'*a*::*order multiset*)
⟨*proof*⟩

theorem *mult-less-asym*:
M < *N* ==> (¬ *P* ==> *N* < (*M*::'*a*::*order multiset*)) ==> *P*
⟨*proof*⟩

theorem *mult-le-refl* [*iff*]: *M* <= (*M*::'*a*::*order multiset*)
⟨*proof*⟩

Anti-symmetry.

theorem *mult-le-antisym*:

$M <= N ==> N <= M ==> M = (N::'a::order\ multiset)$
⟨*proof*⟩

Transitivity.

theorem *mult-le-trans*:
$K <= M ==> M <= N ==> K <= (N::'a::order\ multiset)$
⟨*proof*⟩

theorem *mult-less-le*: $(M < N) = (M <= N \wedge M \neq (N::'a::order\ multiset))$
⟨*proof*⟩

instance
⟨*proof*⟩

end

1.4.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in mult1\ r ==> trans\ r ==> (C + B, C + D) \in mult1\ r$
⟨*proof*⟩

lemma *union-less-mono2*: $B < D ==> C + B < C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-less-mono1*: $B < D ==> B + C < D + (C::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-less-mono*:
$A < C ==> B < D ==> A + B < C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-le-mono*:
$A <= C ==> B <= D ==> A + B <= C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
⟨*proof*⟩

instance *multiset* :: (*order*) *pordered-ab-semigroup-add*
⟨*proof*⟩

1.5 Link with lists

primrec *multiset-of* :: *'a list* ⇒ *'a multiset* **where**
multiset-of [] = {#} |
multiset-of (*a* # *x*) = *multiset-of x* + {# *a* #}

lemma *multiset-of-zero-iff*[*simp*]: (*multiset-of x* = {#}) = (*x* = [])
⟨*proof*⟩

lemma *multiset-of-zero-iff-right*[*simp*]: ({#} = *multiset-of x*) = (*x* = [])
⟨*proof*⟩

lemma *set-of-multiset-of*[*simp*]: *set-of*(*multiset-of x*) = *set x*
⟨*proof*⟩

lemma *mem-set-multiset-eq*: *x* ∈ *set xs* = (*x* :# *multiset-of xs*)
⟨*proof*⟩

lemma *multiset-of-append* [*simp*]:
multiset-of (*xs* @ *ys*) = *multiset-of xs* + *multiset-of ys*
⟨*proof*⟩

lemma *surj-multiset-of*: *surj multiset-of*
⟨*proof*⟩

lemma *set-count-greater-0*: *set x* = {*a*. *count* (*multiset-of x*) *a* > *0*}
⟨*proof*⟩

lemma *distinct-count-atmost-1*:
distinct x = (! *a*. *count* (*multiset-of x*) *a* = (*if a* ∈ *set x then 1 else 0*))
⟨*proof*⟩

lemma *multiset-of-eq-setD*:
multiset-of xs = *multiset-of ys* ⟹ *set xs* = *set ys*
⟨*proof*⟩

lemma *set-eq-iff-multiset-of-eq-distinct*:
distinct x ⟹ *distinct y* ⟹
(*set x* = *set y*) = (*multiset-of x* = *multiset-of y*)
⟨*proof*⟩

lemma *set-eq-iff-multiset-of-remdups-eq*:
(*set x* = *set y*) = (*multiset-of* (*remdups x*) = *multiset-of* (*remdups y*))
⟨*proof*⟩

lemma *multiset-of-compl-union* [*simp*]:
multiset-of [*x*←*xs*. *P x*] + *multiset-of* [*x*←*xs*. ¬*P x*] = *multiset-of xs*
⟨*proof*⟩

lemma *count-filter*:

count (*multiset-of xs*) *x* = *length* [*y* ← *xs*. *y* = *x*]
⟨*proof*⟩

lemma *nth-mem-multiset-of*: *i* < *length ls* ⟹ (*ls* ! *i*) :# *multiset-of ls*
⟨*proof*⟩

lemma *multiset-of-remove1*: *multiset-of* (*remove1 a xs*) = *multiset-of xs* − {#*a*#}
⟨*proof*⟩

lemma *multiset-of-eq-length*:
assumes *multiset-of xs* = *multiset-of ys*
shows *length xs* = *length ys*
⟨*proof*⟩

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

lemma *properties-for-sort*:
multiset-of ys = *multiset-of xs* ⟹ *sorted ys* ⟹ *sort xs* = *ys*
⟨*proof*⟩

1.6 Pointwise ordering induced by count

definition
mset-le :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** ≤# *50*) **where**
(*A* ≤# *B*) = (∀ *a*. *count A a* ≤ *count B a*)

definition
mset-less :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** <# *50*) **where**
(*A* <# *B*) = (*A* ≤# *B* ∧ *A* ≠ *B*)

notation *mset-le* (**infix** ⊆# *50*)
notation *mset-less* (**infix** ⊂# *50*)

lemma *mset-le-refl*[*simp*]: *A* ≤# *A*
⟨*proof*⟩

lemma *mset-le-trans*: *A* ≤# *B* ⟹ *B* ≤# *C* ⟹ *A* ≤# *C*
⟨*proof*⟩

lemma *mset-le-antisym*: *A* ≤# *B* ⟹ *B* ≤# *A* ⟹ *A* = *B*
⟨*proof*⟩

lemma *mset-le-exists-conv*: (*A* ≤# *B*) = (∃ *C*. *B* = *A* + *C*)
⟨*proof*⟩

lemma *mset-le-mono-add-right-cancel*[*simp*]: (*A* + *C* ≤# *B* + *C*) = (*A* ≤# *B*)
⟨*proof*⟩

lemma *mset-le-mono-add-left-cancel*[*simp*]: (*C* + *A* ≤# *C* + *B*) = (*A* ≤# *B*)

⟨*proof*⟩

lemma *mset-le-mono-add*: ⟦ $A \leq\# B$; $C \leq\# D$ ⟧ ⟹ $A + C \leq\# B + D$
⟨*proof*⟩

lemma *mset-le-add-left*[*simp*]: $A \leq\# A + B$
⟨*proof*⟩

lemma *mset-le-add-right*[*simp*]: $B \leq\# A + B$
⟨*proof*⟩

lemma *mset-le-single*: $a :\# B \Longrightarrow \{\#a\#\} \leq\# B$
⟨*proof*⟩

lemma *multiset-diff-union-assoc*: $C \leq\# B \Longrightarrow A + B - C = A + (B - C)$
⟨*proof*⟩

lemma *mset-le-multiset-union-diff-commute*:
assumes $B \leq\# A$
shows $A - B + C = A + C - B$
⟨*proof*⟩

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups xs*) $\leq\#$ *multiset-of xs*
⟨*proof*⟩

lemma *multiset-of-update*:
$i <$ *length ls* ⟹ *multiset-of* ($ls[i := v]$) = *multiset-of ls* − {#*ls* ! *i*#} + {#*v*#}
⟨*proof*⟩

lemma *multiset-of-swap*:
$i <$ *length ls* ⟹ $j <$ *length ls* ⟹
multiset-of ($ls[j :=$ *ls* ! *i*, *i* := *ls* ! *j*]) = *multiset-of ls*
⟨*proof*⟩

interpretation *mset-order*: *order* [*op* $\leq\#$ *op* $<\#$]
⟨*proof*⟩

interpretation *mset-order-cancel-semigroup*:
pordered-cancel-ab-semigroup-add [*op* + *op* $\leq\#$ *op* $<\#$]
⟨*proof*⟩

interpretation *mset-order-semigroup-cancel*:
pordered-ab-semigroup-add-imp-le [*op* + *op* $\leq\#$ *op* $<\#$]
⟨*proof*⟩

lemma *mset-lessD*: $A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
⟨*proof*⟩

lemma *mset-leD*: $A \subseteq\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
⟨*proof*⟩

lemma *mset-less-insertD*: $(A + \{\#x\#\} \subset\# B) \Longrightarrow (x \in\# B \wedge A \subset\# B)$
⟨*proof*⟩

lemma *mset-le-insertD*: $(A + \{\#x\#\} \subseteq\# B) \Longrightarrow (x \in\# B \wedge A \subseteq\# B)$
⟨*proof*⟩

lemma *mset-less-of-empty*[*simp*]: $A \subset\# \{\#\} = \mathit{False}$
⟨*proof*⟩

lemma *multi-psub-of-add-self*[*simp*]: $A \subset\# A + \{\#x\#\}$
⟨*proof*⟩

lemma *multi-psub-self*[*simp*]: $A \subset\# A = \mathit{False}$
⟨*proof*⟩

lemma *mset-less-add-bothsides*:
$T + \{\#x\#\} \subset\# S + \{\#x\#\} \Longrightarrow T \subset\# S$
⟨*proof*⟩

lemma *mset-less-empty-nonempty*: $(\{\#\} \subset\# S) = (S \neq \{\#\})$
⟨*proof*⟩

lemma *mset-less-size*: $A \subset\# B \Longrightarrow \mathit{size}\ A < \mathit{size}\ B$
⟨*proof*⟩

lemmas *mset-less-trans* = *mset-order.less-eq-less.less-trans*

lemma *mset-less-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
⟨*proof*⟩

1.7 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition
mset-less-rel :: (*'a multiset* × *'a multiset*) *set* **where**
mset-less-rel = {(*A*,*B*). $A \subset\# B$}

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$ **and** $b \neq c$
shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
⟨*proof*⟩

lemma *wf-mset-less-rel*: *wf mset-less-rel*
⟨*proof*⟩

The induction rules:

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall A.\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
⟨*proof*⟩

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
  and *empty*: $P\ \{\#\}$
  and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (F + \{\#a\#\})$
shows $P\ F$
⟨*proof*⟩

A consequence: Extensionality.

lemma *multi-count-eq*: $(\forall x.\ count\ A\ x = count\ B\ x) = (A = B)$
⟨*proof*⟩

lemmas *multi-count-ext* = *multi-count-eq* [*THEN iffD1*, *rule-format*]

1.8 The fold combinator

The intended behaviour is *fold-mset* $f\ z\ \{\#x_1, ..., x_n\#\} = f\ x_1\ (\ldots\ (f\ x_n\ z)\ldots)$ if f is associative-commutative.

The graph of *fold-mset*, z: the start element, f: folding function, A: the multiset, y: the result.

inductive
  fold-msetG :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b \Rightarrow bool$
  for $f :: 'a \Rightarrow 'b \Rightarrow 'b$
  and $z :: 'b$
where
  emptyI [*intro*]: *fold-msetG f z* $\{\#\}$ *z*
| *insertI* [*intro*]: *fold-msetG f z A y* $\Longrightarrow$ *fold-msetG f z* $(A + \{\#x\#\})$ $(f\ x\ y)$

inductive-cases *empty-fold-msetGE* [*elim*!]: *fold-msetG f z* $\{\#\}$ *x*
inductive-cases *insert-fold-msetGE*: *fold-msetG f z* $(A + \{\#\})$ *y*

definition
  fold-mset :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b$ **where**
  fold-mset f z A = $(THE\ x.$ *fold-msetG f z A x*$)$

lemma *Diff1-fold-msetG*:
  fold-msetG f z $(A - \{\#x\#\})$ *y* $\Longrightarrow$ $x \in\# A \Longrightarrow$ *fold-msetG f z A* $(f\ x\ y)$
⟨*proof*⟩

lemma *fold-msetG-nonempty*: $\exists x.$ *fold-msetG f z A x*
⟨*proof*⟩

lemma *fold-mset-empty*[*simp*]: *fold-mset f z* $\{\#\} = z$
⟨*proof*⟩

locale *left-commutative* =
fixes *f* :: $'a \Rightarrow 'b \Rightarrow 'b$
assumes *left-commute*: *f x* (*f y z*) = *f y* (*f x z*)
begin

lemma *fold-msetG-determ*:
fold-msetG f z A x $\Longrightarrow$ *fold-msetG f z A y* $\Longrightarrow$ $y = x$
⟨*proof*⟩

lemma *fold-mset-insert-aux*:
(*fold-msetG f z* (*A* + $\{\#x\#\}$) *v*) =
($\exists$ *y*. *fold-msetG f z A y* $\wedge$ *v* = *f x y*)
⟨*proof*⟩

lemma *fold-mset-equality*: *fold-msetG f z A y* $\Longrightarrow$ *fold-mset f z A* = *y*
⟨*proof*⟩

lemma *fold-mset-insert*:
fold-mset f z (*A* + $\{\#x\#\}$) = *f x* (*fold-mset f z A*)
⟨*proof*⟩

lemma *fold-mset-insert-idem*:
fold-mset f z (*A* + $\{\#a\#\}$) = *f a* (*fold-mset f z A*)
⟨*proof*⟩

lemma *fold-mset-commute*: *f x* (*fold-mset f z A*) = *fold-mset f* (*f x z*) *A*
⟨*proof*⟩

lemma *fold-mset-single* [*simp*]: *fold-mset f z* $\{\#x\#\} = f\ x\ z$
⟨*proof*⟩

lemma *fold-mset-union* [*simp*]:
fold-mset f z (*A*+*B*) = *fold-mset f* (*fold-mset f z A*) *B*
⟨*proof*⟩

lemma *fold-mset-fusion*:
includes *left-commutative g*
shows ($\bigwedge x\ y$. *h* (*g x y*) = *f x* (*h y*)) $\Longrightarrow$ *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A*
⟨*proof*⟩

lemma *fold-mset-rec*:
assumes *a* $\in\#$ *A*
shows *fold-mset f z A* = *f a* (*fold-mset f z* (*A* − $\{\#a\#\}$))
⟨*proof*⟩

end

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like *fold-mset F z* $\{\#\} = z$ where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

1.9 Image

definition [*code func del*]: *image-mset f* == *fold-mset* (*op* + *o single o f*) $\{\#\}$

interpretation *image-left-comm*: *left-commutative* [*op* + *o single o f*]
⟨*proof*⟩

lemma *image-mset-empty* [*simp*, *code func*]: *image-mset f* $\{\#\} = \{\#\}$
⟨*proof*⟩

lemma *image-mset-single* [*simp*, *code func*]: *image-mset f* $\{\#x\#\} = \{\#f\ x\#\}$
⟨*proof*⟩

lemma *image-mset-insert*:
image-mset f $(M + \{\#a\#\}) =$ *image-mset f M* $+ \{\#f\ a\#\}$
⟨*proof*⟩

lemma *image-mset-union*[*simp*, *code func*]:
image-mset f $(M+N) =$ *image-mset f M* + *image-mset f N*
⟨*proof*⟩

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
⟨*proof*⟩

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* $= \{\#\} \longleftrightarrow M = \{\#\}$
⟨*proof*⟩

syntax
comprehension1-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset*
(($\{\#$-/. - :# -$\#\}$))
translations
$\{\#e.\ x{:}\#M\#\}$ == *CONST image-mset* (%*x*. *e*) *M*

syntax
comprehension2-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset*
(($\{\#$-/ | - :# -./ -$\#\}$))
translations
$\{\#e \mid x{:}\#M.\ P\#\}$ => $\{\#e.\ x :\#\ \{\#\ x{:}\#M.\ P\#\}\#\}$

This allows to write not just filters like $\{\#\ x :\#\ M.\ x < c\#\}$ but also images like $\{\#x + x.\ x :\#\ M\#\}$ and $\{\#x+x|x:\#M.\ x<c\#\}$, where the latter is currently displayed as $\{\#x + x.\ x :\#\ \{\#\ x :\#\ M.\ x < c\#\}\#\}$.

end

theory *LProd*
imports *Multiset*
begin

inductive-set
  lprod :: $('a * 'a)\ set \Rightarrow ('a\ list * 'a\ list)\ set$
  for R :: $('a * 'a)\ set$
where
  lprod-single[*intro*!]: $(a,\ b) \in R \Longrightarrow ([a],\ [b]) \in lprod\ R$
| *lprod-list*[*intro*!]: $(ah@at,\ bh@bt) \in lprod\ R \Longrightarrow (a,b) \in R \vee a = b \Longrightarrow (ah@a\#at,\ bh@b\#bt) \in lprod\ R$

lemma $(as,bs) \in lprod\ R \Longrightarrow length\ as = length\ bs$
  ⟨*proof*⟩

lemma $(as,\ bs) \in lprod\ R \Longrightarrow 1 \leq length\ as \wedge 1 \leq length\ bs$
  ⟨*proof*⟩

lemma *lprod-subset-elem*: $(as,\ bs) \in lprod\ S \Longrightarrow S \subseteq R \Longrightarrow (as,\ bs) \in lprod\ R$
  ⟨*proof*⟩

lemma *lprod-subset*: $S \subseteq R \Longrightarrow lprod\ S \subseteq lprod\ R$
  ⟨*proof*⟩

lemma *lprod-implies-mult*: $(as,\ bs) \in lprod\ R \Longrightarrow trans\ R \Longrightarrow (multiset\text{-}of\ as,\ multiset\text{-}of\ bs) \in mult\ R$
⟨*proof*⟩

lemma *wf-lprod*[*recdef-wf*,*simp*,*intro*]:
  assumes *wf-R*: *wf R*
  shows *wf* (*lprod R*)
⟨*proof*⟩

constdefs
  gprod-2-2 :: $('a * 'a)\ set \Rightarrow (('a * 'a) * ('a * 'a))\ set$
  gprod-2-2 R ≡ $\{\ ((a,b),\ (c,d))\ .\ (a = c \wedge (b,d) \in R) \vee (b = d \wedge (a,c) \in R)\ \}$
  gprod-2-1 :: $('a * 'a)\ set \Rightarrow (('a * 'a) * ('a * 'a))\ set$
  gprod-2-1 R ≡ $\{\ ((a,b),\ (c,d))\ .\ (a = d \wedge (b,c) \in R) \vee (b = c \wedge (a,d) \in R)\ \}$

lemma *lprod-2-3*: $(a,\ b) \in R \Longrightarrow ([a,\ c],\ [b,\ c]) \in lprod\ R$
  ⟨*proof*⟩

lemma *lprod-2-4*: $(a,\ b) \in R \Longrightarrow ([c,\ a],\ [c,\ b]) \in lprod\ R$
⟨*proof*⟩

lemma *lprod-2-1*: $(a,\ b) \in R \Longrightarrow ([c,\ a],\ [b,\ c]) \in lprod\ R$
⟨*proof*⟩

lemma *lprod-2-2*: $(a,\ b) \in R \Longrightarrow ([a,\ c],\ [c,\ b]) \in lprod\ R$
⟨*proof*⟩

lemma [*recdef-wf*, *simp*, *intro*]:
assumes *wfR*: *wf R* **shows** *wf* (*gprod-2-1 R*)
⟨*proof*⟩

lemma [*recdef-wf*, *simp*, *intro*]:
assumes *wfR*: *wf R* **shows** *wf* (*gprod-2-2 R*)
⟨*proof*⟩

lemma *lprod-3-1*: **assumes** $(x',\ x) \in R$ **shows** $([y,\ z,\ x'],\ [x,\ y,\ z]) \in lprod\ R$
⟨*proof*⟩

lemma *lprod-3-2*: **assumes** $(z',z) \in R$ **shows** $([z',\ x,\ y],\ [x,y,z]) \in lprod\ R$
⟨*proof*⟩

lemma *lprod-3-3*: **assumes** *xr*: $(xr,\ x) \in R$ **shows** $([xr,\ y,\ z],\ [x,\ y,\ z]) \in lprod\ R$
⟨*proof*⟩

lemma *lprod-3-4*: **assumes** *yr*: $(yr,\ y) \in R$ **shows** $([x,\ yr,\ z],\ [x,\ y,\ z]) \in lprod\ R$
⟨*proof*⟩

lemma *lprod-3-5*: **assumes** *zr*: $(zr,\ z) \in R$ **shows** $([x,\ y,\ zr],\ [x,\ y,\ z]) \in lprod\ R$
⟨*proof*⟩

lemma *lprod-3-6*: **assumes** *y'*: $(y',\ y) \in R$ **shows** $([x,\ z,\ y'],\ [x,\ y,\ z]) \in lprod\ R$
⟨*proof*⟩

lemma *lprod-3-7*: **assumes** *z'*: $(z',z) \in R$ **shows** $([x,\ z',\ y],\ [x,\ y,\ z]) \in lprod\ R$
⟨*proof*⟩

constdefs
perm :: $('a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow bool$
perm f A ≡ *inj-on f A* ∧ *f* ‘ *A* = *A*

lemma $((as,bs) \in lprod\ R) =$
$(\exists\ f.\ perm\ f\ \{0\ ..<\ (length\ as)\} \wedge$
$(\forall\ j.\ j < length\ as \longrightarrow ((nth\ as\ j,\ nth\ bs\ (f\ j)) \in R \vee (nth\ as\ j = nth\ bs\ (f\ j))))$
∧
$(\exists\ i.\ i < length\ as \wedge (nth\ as\ i,\ nth\ bs\ (f\ i)) \in R))$
⟨*proof*⟩

lemma *trans R* $\Longrightarrow$ (*ah*@*a*#*at*, *bh*@*b*#*bt*) $\in$ *lprod R* $\Longrightarrow$ (*b*, *a*) $\in$ *R* $\vee$ *a* = *b* $\Longrightarrow$ (*ah*@*at*, *bh*@*bt*) $\in$ *lprod R*
⟨*proof*⟩

end

theory *MainZF*
imports *Zet LProd*
begin
end

theory *Games*
imports *MainZF*
begin

constdefs
fixgames :: *ZF set* ⇒ *ZF set*
fixgames A ≡ { *Opair l r* | *l r*. *explode l* ⊆ *A* & *explode r* ⊆ *A*}
games-lfp :: *ZF set*
games-lfp ≡ *lfp fixgames*
games-gfp :: *ZF set*
games-gfp ≡ *gfp fixgames*

lemma *mono-fixgames*: *mono* (*fixgames*)
⟨*proof*⟩

lemma *games-lfp-unfold*: *games-lfp* = *fixgames games-lfp*
⟨*proof*⟩

lemma *games-gfp-unfold*: *games-gfp* = *fixgames games-gfp*
⟨*proof*⟩

lemma *games-lfp-nonempty*: *Opair Empty Empty* ∈ *games-lfp*
⟨*proof*⟩

constdefs
left-option :: *ZF* ⇒ *ZF* ⇒ *bool*
left-option g opt ≡ (*Elem opt* (*Fst g*))
right-option :: *ZF* ⇒ *ZF* ⇒ *bool*
right-option g opt ≡ (*Elem opt* (*Snd g*))
is-option-of :: (*ZF* ∗ *ZF*) *set*
is-option-of ≡ { (*opt*, *g*) | *opt g*. *g* ∈ *games-gfp* ∧ (*left-option g opt* ∨ *right-option g opt*) }

lemma *games-lfp-subset-gfp*: *games-lfp* ⊆ *games-gfp*

⟨*proof*⟩

lemma *games-option-stable*:
  assumes *fixgames*: *games* = *fixgames games*
  and *g*: *g* ∈ *games*
  and *opt*: *left-option g opt* ∨ *right-option g opt*
  shows *opt* ∈ *games*
⟨*proof*⟩

lemma *option2elem*: (*opt*,*g*) ∈ *is-option-of* ⟹ ∃ *u v*. *Elem opt u* ∧ *Elem u v* ∧ *Elem v g*
  ⟨*proof*⟩

lemma *is-option-of-subset-is-Elem-of*: *is-option-of* ⊆ (*is-Elem-of*^+)
⟨*proof*⟩

lemma *wfzf-is-option-of*: *wfzf is-option-of*
⟨*proof*⟩

lemma *games-gfp-imp-lfp*: *g* ∈ *games-gfp* ⟶ *g* ∈ *games-lfp*
⟨*proof*⟩

theorem *games-lfp-eq-gfp*: *games-lfp* = *games-gfp*
  ⟨*proof*⟩

theorem *unique-games*: (*g* = *fixgames g*) = (*g* = *games-lfp*)
⟨*proof*⟩

lemma *games-lfp-option-stable*:
  assumes *g*: *g* ∈ *games-lfp*
  and *opt*: *left-option g opt* ∨ *right-option g opt*
  shows *opt* ∈ *games-lfp*
  ⟨*proof*⟩

lemma *is-option-of-imp-games*:
  assumes *hyp*: (*opt*, *g*) ∈ *is-option-of*
  shows *opt* ∈ *games-lfp* ∧ *g* ∈ *games-lfp*
⟨*proof*⟩

lemma *games-lfp-represent*: *x* ∈ *games-lfp* ⟹ ∃ *l r*. *x* = *Opair l r*
  ⟨*proof*⟩

typedef *game* = *games-lfp*
  ⟨*proof*⟩

constdefs
  left-options :: *game* ⇒ *game zet*
  left-options g ≡ *zimage Abs-game* (*zexplode* (*Fst* (*Rep-game g*)))
  right-options :: *game* ⇒ *game zet*

right-options g ≡ *zimage Abs-game* (*zexplode* (*Snd* (*Rep-game g*)))
options :: *game* ⇒ *game zet*
options g ≡ *zunion* (*left-options g*) (*right-options g*)
Game :: *game zet* ⇒ *game zet* ⇒ *game*
Game L R ≡ *Abs-game* (*Opair* (*zimplode* (*zimage Rep-game L*)) (*zimplode* (*zimage Rep-game R*)))

lemma *Repl-Rep-game-Abs-game*: ∀ *e*. *Elem e z* ⟶ *e* ∈ *games-lfp* ⟹ *Repl z* (*Rep-game o Abs-game*) = *z*
⟨*proof*⟩

lemma *game-split*: *g* = *Game* (*left-options g*) (*right-options g*)
⟨*proof*⟩

lemma *Opair-in-games-lfp*:
assumes *l*: *explode l* ⊆ *games-lfp*
and *r*: *explode r* ⊆ *games-lfp*
shows *Opair l r* ∈ *games-lfp*
⟨*proof*⟩

lemma *left-options*[*simp*]: *left-options* (*Game l r*) = *l*
⟨*proof*⟩

lemma *right-options*[*simp*]: *right-options* (*Game l r*) = *r*
⟨*proof*⟩

lemma *Game-ext*: (*Game l1 r1* = *Game l2 r2*) = ((*l1* = *l2*) ∧ (*r1* = *r2*))
⟨*proof*⟩

constdefs
option-of :: (*game* ∗ *game*) *set*
option-of ≡ *image* (λ (*option*, *g*). (*Abs-game option*, *Abs-game g*)) *is-option-of*

lemma *option-to-is-option-of*: ((*option*, *g*) ∈ *option-of*) = ((*Rep-game option*, *Rep-game g*) ∈ *is-option-of*)
⟨*proof*⟩

lemma *wf-is-option-of*: *wf is-option-of*
⟨*proof*⟩

lemma *wf-option-of*[*recdef-wf*, *simp*, *intro*]: *wf option-of*
⟨*proof*⟩

lemma *right-option-is-option*[*simp*, *intro*]: *zin x* (*right-options g*) ⟹ *zin x* (*options g*)
⟨*proof*⟩

lemma *left-option-is-option*[*simp*, *intro*]: *zin x* (*left-options g*) ⟹ *zin x* (*options g*)

⟨*proof*⟩

lemma *zin-options*[*simp*, *intro*]: *zin x* (*options g*) ⟹ (*x*, *g*) ∈ *option-of*
⟨*proof*⟩

consts
neg-game :: *game* ⇒ *game*

recdef *neg-game option-of*
neg-game g = *Game* (*zimage neg-game* (*right-options g*)) (*zimage neg-game* (*left-options g*))

declare *neg-game.simps*[*simp del*]

lemma *neg-game* (*neg-game g*) = *g*
⟨*proof*⟩

consts
ge-game :: (*game* ∗ *game*) ⇒ *bool*

recdef *ge-game* (*gprod-2-1 option-of*)
ge-game (*G*, *H*) = (∀ *x*. *if zin x* (*right-options G*) *then* (
if zin x (*left-options H*) *then* ¬ (*ge-game* (*H*, *x*) ∨ (*ge-game* (*x*, *G*)))
else ¬ (*ge-game* (*H*, *x*)))
else (*if zin x* (*left-options H*) *then* ¬ (*ge-game* (*x*, *G*)) *else True*))
(**hints** *simp*: *gprod-2-1-def*)

declare *ge-game.simps* [*simp del*]

lemma *ge-game-eq*: *ge-game* (*G*, *H*) = (∀ *x*. (*zin x* (*right-options G*) ⟶ ¬ *ge-game* (*H*, *x*)) ∧ (*zin x* (*left-options H*) ⟶ ¬ *ge-game* (*x*, *G*)))
⟨*proof*⟩

lemma *ge-game-leftright-refl*[*rule-format*]:
∀ *y*. (*zin y* (*right-options x*) ⟶ ¬ *ge-game* (*x*, *y*)) ∧ (*zin y* (*left-options x*) ⟶ ¬ (*ge-game* (*y*, *x*))) ∧ *ge-game* (*x*, *x*)
⟨*proof*⟩

lemma *ge-game-refl*: *ge-game* (*x*,*x*) ⟨*proof*⟩

lemma ∀ *y*. (*zin y* (*right-options x*) ⟶ ¬ *ge-game* (*x*, *y*)) ∧ (*zin y* (*left-options x*) ⟶ ¬ (*ge-game* (*y*, *x*))) ∧ *ge-game* (*x*, *x*)
⟨*proof*⟩

constdefs
eq-game :: *game* ⇒ *game* ⇒ *bool*
eq-game G H ≡ *ge-game* (*G*, *H*) ∧ *ge-game* (*H*, *G*)

lemma *eq-game-sym*: (*eq-game G H*) = (*eq-game H G*)
⟨*proof*⟩

lemma *eq-game-refl*: *eq-game G G*
⟨*proof*⟩

lemma *induct-game*: (⋀*x*. ∀ *y*. (*y*, *x*) ∈ *lprod option-of* ⟶ *P y* ⟹ *P x*) ⟹ *P a*
⟨*proof*⟩

lemma *ge-game-trans*:
assumes *ge-game* (*x*, *y*) *ge-game* (*y*, *z*)
shows *ge-game* (*x*, *z*)
⟨*proof*⟩

lemma *eq-game-trans*: *eq-game a b* ⟹ *eq-game b c* ⟹ *eq-game a c*
⟨*proof*⟩

constdefs
zero-game :: *game*
zero-game ≡ *Game zempty zempty*

consts
plus-game :: *game* ∗ *game* ⇒ *game*

recdef *plus-game gprod-2-2 option-of*
plus-game (*G*, *H*) = *Game* (*zunion* (*zimage* (λ *g*. *plus-game* (*g*, *H*)) (*left-options G*))
(*zimage* (λ *h*. *plus-game* (*G*, *h*)) (*left-options H*)))
(*zunion* (*zimage* (λ *g*. *plus-game* (*g*, *H*)) (*right-options G*))
(*zimage* (λ *h*. *plus-game* (*G*, *h*)) (*right-options H*)))
(**hints** *simp add*: *gprod-2-2-def*)

declare *plus-game.simps*[*simp del*]

lemma *plus-game-comm*: *plus-game* (*G*, *H*) = *plus-game* (*H*, *G*)
⟨*proof*⟩

lemma *game-ext-eq*: (*G* = *H*) = (*left-options G* = *left-options H* ∧ *right-options G* = *right-options H*)
⟨*proof*⟩

lemma *left-zero-game*[*simp*]: *left-options* (*zero-game*) = *zempty*
⟨*proof*⟩

lemma *right-zero-game*[*simp*]: *right-options* (*zero-game*) = *zempty*
⟨*proof*⟩

lemma *plus-game-zero-right*[*simp*]: *plus-game* (*G*, *zero-game*) = *G*
⟨*proof*⟩

lemma *plus-game-zero-left*: *plus-game* (*zero-game*, *G*) = *G*
⟨*proof*⟩

lemma *left-imp-options*[*simp*]: *zin opt* (*left-options g*) ⟹ *zin opt* (*options g*)
⟨*proof*⟩

lemma *right-imp-options*[*simp*]: *zin opt* (*right-options g*) ⟹ *zin opt* (*options g*)
⟨*proof*⟩

lemma *left-options-plus*:
left-options (*plus-game* (*u*, *v*)) = *zunion* (*zimage* (λ*g*. *plus-game* (*g*, *v*)) (*left-options u*)) (*zimage* (λ*h*. *plus-game* (*u*, *h*)) (*left-options v*))
⟨*proof*⟩

lemma *right-options-plus*:
right-options (*plus-game* (*u*, *v*)) = *zunion* (*zimage* (λ*g*. *plus-game* (*g*, *v*)) (*right-options u*)) (*zimage* (λ*h*. *plus-game* (*u*, *h*)) (*right-options v*))
⟨*proof*⟩

lemma *left-options-neg*: *left-options* (*neg-game u*) = *zimage neg-game* (*right-options u*)
⟨*proof*⟩

lemma *right-options-neg*: *right-options* (*neg-game u*) = *zimage neg-game* (*left-options u*)
⟨*proof*⟩

lemma *plus-game-assoc*: *plus-game* (*plus-game* (*F*, *G*), *H*) = *plus-game* (*F*, *plus-game* (*G*, *H*))
⟨*proof*⟩

lemma *neg-plus-game*: *neg-game* (*plus-game* (*G*, *H*)) = *plus-game*(*neg-game G*, *neg-game H*)
⟨*proof*⟩

lemma *eq-game-plus-inverse*: *eq-game* (*plus-game* (*x*, *neg-game x*)) *zero-game*
⟨*proof*⟩

lemma *ge-plus-game-left*: *ge-game* (*y*,*z*) = *ge-game*(*plus-game* (*x*, *y*), *plus-game* (*x*, *z*))
⟨*proof*⟩

lemma *ge-plus-game-right*: *ge-game* (*y*,*z*) = *ge-game*(*plus-game* (*y*, *x*), *plus-game* (*z*, *x*))
⟨*proof*⟩

lemma *ge-neg-game*: *ge-game* (*neg-game x*, *neg-game y*) = *ge-game* (*y*, *x*)
⟨*proof*⟩

constdefs
eq-game-rel :: (*game* ∗ *game*) *set*
eq-game-rel ≡ { (*p*, *q*) . *eq-game p q* }

typedef *Pg* = *UNIV*//*eq-game-rel*
⟨*proof*⟩

lemma *equiv-eq-game*[*simp*]: *equiv UNIV eq-game-rel*
⟨*proof*⟩

instantiation *Pg* :: {*ord*, *zero*, *plus*, *minus*, *uminus*}
begin

definition
Pg-zero-def: *0* = *Abs-Pg* (*eq-game-rel* `` {*zero-game*})

definition
Pg-le-def: *G* ≤ *H* ⟷ (∃ *g h*. *g* ∈ *Rep-Pg G* ∧ *h* ∈ *Rep-Pg H* ∧ *ge-game* (*h*, *g*))

definition
Pg-less-def: *G* < *H* ⟷ *G* ≤ *H* ∧ *G* ≠ (*H*::*Pg*)

definition
Pg-minus-def: − *G* = *contents* (⋃ *g* ∈ *Rep-Pg G*. {*Abs-Pg* (*eq-game-rel* `` {*neg-game g*})})

definition
Pg-plus-def: *G* + *H* = *contents* (⋃ *g* ∈ *Rep-Pg G*. ⋃ *h* ∈ *Rep-Pg H*. {*Abs-Pg* (*eq-game-rel* `` {*plus-game* (*g*,*h*)})})

definition
Pg-diff-def: *G* − *H* = *G* + (− (*H*::*Pg*))

instance ⟨*proof*⟩

end

lemma *Rep-Abs-eq-Pg*[*simp*]: *Rep-Pg* (*Abs-Pg* (*eq-game-rel* `` {*g*})) = *eq-game-rel* `` {*g*}
⟨*proof*⟩

lemma *char-Pg-le*[*simp*]: (*Abs-Pg* (*eq-game-rel* `` {*g*}) ≤ *Abs-Pg* (*eq-game-rel* `` {*h*})) = (*ge-game* (*h*, *g*))
⟨*proof*⟩

lemma *char-Pg-eq*[*simp*]: (*Abs-Pg* (*eq-game-rel* `` {*g*}) = *Abs-Pg* (*eq-game-rel* `` {*h*})) = (*eq-game g h*)
⟨*proof*⟩

lemma *char-Pg-plus*[*simp*]: *Abs-Pg* (*eq-game-rel* `` {*g*}) + *Abs-Pg* (*eq-game-rel* `` {*h*}) = *Abs-Pg* (*eq-game-rel* `` {*plus-game* (*g*, *h*)})
⟨*proof*⟩

lemma *char-Pg-minus*[*simp*]: − *Abs-Pg* (*eq-game-rel* `` {*g*}) = *Abs-Pg* (*eq-game-rel* `` {*neg-game g*})
⟨*proof*⟩

lemma *eq-Abs-Pg*[*rule-format*, *cases type*: *Pg*]: (∀ *g*. *z* = *Abs-Pg* (*eq-game-rel* `` {*g*}) ⟶ *P*) ⟶ *P*
⟨*proof*⟩

instance *Pg* :: *pordered-ab-group-add*
⟨*proof*⟩

end