ZF

Steven Obua

June 21, 2010

theory *HOLZF*
imports *Main*
begin

typedecl *ZF*

axiomatization
  Empty :: *ZF* **and**
  Elem :: *ZF* ⇒ *ZF* ⇒ *bool* **and**
  Sum :: *ZF* ⇒ *ZF* **and**
  Power :: *ZF* ⇒ *ZF* **and**
  Repl :: *ZF* ⇒ (*ZF* ⇒ *ZF*) ⇒ *ZF* **and**
  Inf :: *ZF*

definition *Upair* :: *ZF* ⇒ *ZF* ⇒ *ZF* **where**
  Upair a b == *Repl* (*Power* (*Power Empty*)) (% *x*. *if x* = *Empty then a else b*)

definition *Singleton*:: *ZF* ⇒ *ZF* **where**
  Singleton x == *Upair x x*

definition *union* :: *ZF* ⇒ *ZF* ⇒ *ZF* **where**
  union A B == *Sum* (*Upair A B*)

definition *SucNat*:: *ZF* ⇒ *ZF* **where**
  SucNat x == *union x* (*Singleton x*)

definition *subset* :: *ZF* ⇒ *ZF* ⇒ *bool* **where**
  subset A B == ! *x*. *Elem x A* ⟶ *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* ≠ *Empty* ⟶ (*?* *x*. *Elem x A* & (! *y*. *Elem y x* ⟶ *Not* (*Elem y*

A)))
Infinity: *Elem Empty Inf* & (! *x*. *Elem x Inf* ⟶ *Elem* (*SucNat x*) *Inf*)

definition *Sep* :: *ZF* ⇒ (*ZF* ⇒ *bool*) ⇒ *ZF* **where**
Sep A p == (*if* (!*x*. *Elem x A* ⟶ *Not* (*p x*)) *then Empty else*
(*let z* = (ϵ *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* | *x* = *b*)
⟨*proof*⟩

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

definition *Opair* :: *ZF* ⇒ *ZF* ⇒ *ZF* **where**
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*) | (*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*⟩

definition *Replacement* :: *ZF* ⇒ (*ZF* ⇒ *ZF option*) ⇒ *ZF* **where**
Replacement A f == *Repl* (*Sep A* (% *a*. *f a* ≠ *None*)) (*the o f*)

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

definition *Fst* :: *ZF* ⇒ *ZF* **where**
Fst q == *SOME x*. *? y*. *q* = *Opair x y*

definition *Snd* :: *ZF* ⇒ *ZF* **where**
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*⟩

definition *isOpair* :: *ZF* ⇒ *bool* **where**
isOpair q == *? x y*. *q* = *Opair x y*

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

lemma *FstSnd*: *isOpair x* ⟹ *Opair* (*Fst x*) (*Snd x*) = *x*
⟨*proof*⟩

definition *CartProd* :: *ZF* ⇒ *ZF* ⇒ *ZF* **where**
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*⟩

definition *explode* :: *ZF* ⇒ *ZF set* **where**
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*⟩

definition *Domain* :: *ZF* ⇒ *ZF* **where**
Domain f == *Replacement f* (% *p*. *if isOpair p then Some* (*Fst p*) *else None*)

definition *Range* :: *ZF* ⇒ *ZF* **where**
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*⟩

definition *Field* :: *ZF* ⇒ *ZF* **where**
Field A == *union* (*Domain A*) (*Range A*)

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

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

definition *Lambda* :: *ZF* ⇒ (*ZF* ⇒ *ZF*) ⇒ *ZF* **where**
Lambda A f == *Repl A* (% *x. Opair x* (*f x*))

lemma *Lambda-app*: *Elem x A* ⟹ (*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* ⟶ *f x* = *g x*))
⟨*proof*⟩

definition *PFun* :: *ZF* ⇒ *ZF* ⇒ *ZF* **where**
PFun A B == *Sep* (*Power* (*CartProd A B*)) *isFun*

definition *Fun* :: *ZF* ⇒ *ZF* ⇒ *ZF* **where**
Fun A B == *Sep* (*PFun A B*) (λ *f. Domain f* = *A*)

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

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

lemma *Fun-implies-PFun*[*simp*]: *Elem f* (*Fun U V*) ⟹ *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*⟩

definition *is-Elem-of* :: *(ZF * ZF) set* **where**
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* ⇒ *ZF*

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

definition *Nat2nat* :: *ZF* ⇒ *nat* **where**
Nat2nat == *inv nat2Nat*

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

definition *Nat* :: *ZF*
where *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*⟩

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

lemma *wf SpecialR*
⟨*proof*⟩

definition *Ext* :: $('a * 'b)\ set \Rightarrow 'b \Rightarrow 'a\ set$ **where**
Ext R y $\equiv \{\ x\ .\ (x,\ y) \in R\ \}$

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

lemma *Ext SpecialR Empty* $\neq$ *explode z*
⟨*proof*⟩

definition *implode* :: *ZF set* $\Rightarrow$ *ZF* **where**
implode == *inv explode*

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

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

definition *regular* :: $(ZF * ZF)\ set \Rightarrow bool$ **where**
regular R == ! *A*. *A* $\neq$ *Empty* $\longrightarrow$ (*? x*. *Elem x A* & (! *y*. (*y*, *x*) $\in$ *R* $\longrightarrow$ *Not* (*Elem y A*)))

definition *set-like* :: $(ZF * ZF)\ set \Rightarrow bool$ **where**
set-like R == ! *y*. *Ext R y* $\in$ *range explode*

definition *wfzf* :: $(ZF * ZF)\ set \Rightarrow bool$ **where**
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*⟩

definition *SeqSum* :: $(nat \Rightarrow ZF) \Rightarrow ZF$ **where**
SeqSum f == *Sum* (*Repl Nat* (*f o Nat2nat*))

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

definition *Ext-ZF* :: $(ZF * ZF)\ set \Rightarrow ZF \Rightarrow ZF$ **where**
Ext-ZF R s == *implode* (*Ext R s*)

lemma *Elem-implode*: *A* $\in$ *range explode* $\Longrightarrow$ *Elem x* (*implode A*) = (*x* $\in$ *A*)
⟨*proof*⟩

lemma *Elem-Ext-ZF*: *set-like R* $\Longrightarrow$ *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*))

definition *Ext-ZF-hull* :: (*ZF* ∗ *ZF*) *set* ⇒ *ZF* ⇒ *ZF* **where**
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* $\Longrightarrow$ *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 $\Longrightarrow$ *x* ∈ (*Ext* (*R*^+) *s*) $\Longrightarrow$ *Elem x* (*Ext-ZF-hull R s*)
⟨*proof*⟩

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

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

lemma *set-like R* $\Longrightarrow$ *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*⟩

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

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

definition *zimage* :: (*'a* ⇒ *'b*) ⇒ *'a zet* ⇒ *'b zet* **where**
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-zet-rep*: *A* ∈ *zet* ⟹ *?* *z* . *g* ‘ *A* = *explode z*
⟨*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*⟩

definition *zimplode* :: *ZF zet* ⇒ *ZF* **where**
zimplode A == *implode* (*Rep-zet A*)

definition *zexplode* :: *ZF* ⇒ *ZF zet* **where**
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*⟩

definition *zunion* :: *'a zet* ⇒ *'a zet* ⇒ *'a zet* **where**
zunion a b ≡ *Abs-zet* ((*Rep-zet a*) ∪ (*Rep-zet b*))

definition *zsubset* :: *'a zet* ⇒ *'a zet* ⇒ *bool* **where**
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*⟩

definition *zempty* :: *'a zet* **where**
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*, *fundef-cong*]: ⟦ *M* = *N*; !! *x*. *zin x N* ⟹ *f x* = *g x* ⟧ ⟹ *zimage f M* = *zimage g N*
⟨*proof*⟩

end

1 (Finite) multisets

theory *Multiset*
imports *Main*
begin

1.1 The type of multisets

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

lemmas *multiset-typedef* = *Abs-multiset-inverse count-inverse count*

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

notation (*xsymbols*)
Melem (**infix** $\in\#$ *50*)

lemma *multiset-ext-iff*:
$M = N \longleftrightarrow (\forall a.\ count\ M\ a = count\ N\ a)$
⟨*proof*⟩

lemma *multiset-ext*:
$(\bigwedge x.\ count\ A\ x = count\ B\ x) \Longrightarrow A = B$
⟨*proof*⟩

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*:
$(\lambda a.\ 0) \in multiset$
⟨*proof*⟩

lemma *only1-in-multiset*:
$(\lambda b.\ if\ b = a\ then\ n\ else\ 0) \in multiset$
⟨*proof*⟩

lemma *union-preserves-multiset*:
$M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
⟨*proof*⟩

lemma *diff-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda a.\ M\ a - N\ a) \in multiset$
⟨*proof*⟩

lemma *MCollect-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
⟨*proof*⟩

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

1.2 Representing multisets

Multiset comprehension

definition *MCollect* :: *'a multiset => ('a => bool) => 'a multiset* **where**
MCollect M P = *Abs-multiset* $(\lambda x.\ if\ P\ x\ then\ count\ M\ x\ else\ 0)$

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

Multiset enumeration

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

definition *Mempty-def*:
0 = *Abs-multiset* (λ*a*. *0*)

abbreviation *Mempty* :: '*a multiset* ({#}) **where**
Mempty ≡ *0*

definition *union-def*:
M + *N* = *Abs-multiset* (λ*a*. *count M a* + *count N a*)

instance ⟨*proof*⟩

end

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

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

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

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

1.3 Basic operations

1.3.1 Union

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

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

1.3.2 Difference

instantiation *multiset* :: (*type*) *minus*
begin

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

instance ⟨*proof*⟩

end

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

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

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

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

lemma *diff-union-cancelL* [*simp*]: $N + M - N = (M::'a\ multiset)$
⟨*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 *diff-right-commute*:
$(M::'a\ multiset) - N - Q = M - Q - N$
⟨*proof*⟩

lemma *diff-add*:
$(M::'a\ multiset) - (N + Q) = M - N - Q$
⟨*proof*⟩

lemma *diff-union-swap*:
$a \neq b \Longrightarrow M - \{\#a\#\} + \{\#b\#\} = M + \{\#b\#\} - \{\#a\#\}$
⟨*proof*⟩

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

1.3.3 Equality of multisets

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

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

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

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

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

lemma *diff-single-trivial*:
$\neg\ x \in\# M \Longrightarrow M - \{\#x\#\} = M$
⟨*proof*⟩

lemma *diff-single-eq-union*:
$x \in\# M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = N + \{\#x\#\}$
⟨*proof*⟩

lemma *union-single-eq-diff*:
$M + \{\#x\#\} = N \Longrightarrow M = N - \{\#x\#\}$
⟨*proof*⟩

lemma *union-single-eq-member*:
$M + \{\#x\#\} = N \Longrightarrow x \in\# N$
⟨*proof*⟩

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

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

lemma *add-eq-conv-diff*:
$M + \{\#a\#\} = N + \{\#b\#\} \longleftrightarrow M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\}$ (**is** *?lhs = ?rhs*)

⟨*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*⟩

1.3.4 Pointwise ordering induced by count

instantiation *multiset* :: (*type*) *ordered-ab-semigroup-add-imp-le*
begin

definition *less-eq-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
mset-le-def: $A \leq B \longleftrightarrow (\forall a.\ count\ A\ a \leq count\ B\ a)$

definition *less-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
mset-less-def: $(A::'a\ multiset) < B \longleftrightarrow A \leq B \wedge A \neq B$

instance ⟨*proof*⟩

end

lemma *mset-less-eqI*:
$(\bigwedge x.\ count\ A\ x \leq count\ B\ x) \Longrightarrow A \leq B$
⟨*proof*⟩

lemma *mset-le-exists-conv*:
$(A::'a\ multiset) \leq B \longleftrightarrow (\exists C.\ B = A + C)$
⟨*proof*⟩

lemma *mset-le-mono-add-right-cancel* [*simp*]:
$(A::'a\ multiset) + C \leq B + C \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *mset-le-mono-add-left-cancel* [*simp*]:
$C + (A::'a\ multiset) \leq C + B \longleftrightarrow A \leq B$
⟨*proof*⟩

lemma *mset-le-mono-add*:
$(A::'a\ multiset) \leq B \Longrightarrow C \leq D \Longrightarrow A + C \leq B + D$
⟨*proof*⟩

lemma *mset-le-add-left* [*simp*]:
$(A::'a\ multiset) \leq A + B$
⟨*proof*⟩

lemma *mset-le-add-right* [*simp*]:
$B \leq (A::'a\ multiset) + B$
⟨*proof*⟩

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

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

lemma *mset-le-multiset-union-diff-commute*:
$B \leq A \Longrightarrow (A::'a\ multiset) - B + C = A + C - B$
⟨*proof*⟩

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

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

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

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

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

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

lemma *multi-psub-self*[*simp*]: $(A::'a\ multiset) < A = False$
⟨*proof*⟩

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

lemma *mset-less-empty-nonempty*:
$\{\#\} < S \longleftrightarrow S \neq \{\#\}$
⟨*proof*⟩

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

1.3.5 Intersection

instantiation *multiset* :: (*type*) *semilattice-inf*
begin

definition *inf-multiset* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow 'a\ multiset$ **where**
multiset-inter-def: *inf-multiset A B* $= A - (A - B)$

instance ⟨*proof*⟩

end

abbreviation *multiset-inter* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* (**infixl** #∩ *70*) **where**
multiset-inter ≡ *inf*

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

lemma *multiset-inter-single*: *a* ≠ *b* ⟹ {#*a*#} #∩ {#*b*#} = {#}
⟨*proof*⟩

lemma *multiset-union-diff-commute*:
assumes *B* #∩ *C* = {#}
shows *A* + *B* − *C* = *A* − *C* + *B*
⟨*proof*⟩

1.3.6 Comprehension (filter)

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

lemma *MCollect-empty* [*simp*]: *MCollect* {#} *P* = {#}
⟨*proof*⟩

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

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

1.3.7 Set of elements

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

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*⟩

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
⟨*proof*⟩

1.3.8 Size

instantiation *multiset* :: (*type*) *size*
begin

definition *size-def*:
size M = *setsum* (*count M*) (*set-of M*)

instance ⟨*proof*⟩

end

lemma *size-empty* [*simp*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
⟨*proof*⟩

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count N*) (*A* ∩ *set-of N*) = *setsum* (*count N*) *A*
⟨*proof*⟩

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

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

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ==> ∃ *a*. *a* :# *M*
⟨*proof*⟩

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

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

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

lemma *mset-less-size*: *(A::'a multiset) < B ==> size A < size B*
⟨*proof*⟩

1.4.1 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* < *B*}

lemma *multiset-add-sub-el-shuffle*:
assumes *c* ∈# *B* **and** *b* ≠ *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*: ⋀*B*. ∀(*A*::'*a multiset*). *A* < *B* ⟶ *P A* ⟹ *P B*
shows *P B*
⟨*proof*⟩

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes *F* ≤ *A*
and *empty*: *P* {#}
and *insert*: ⋀*a F*. *a* ∈# *A* ⟹ *P F* ⟹ *P* (*F* + {#*a*#})
shows *P F*
⟨*proof*⟩

1.5 Alternative representations

1.5.1 Lists

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

lemma *in-multiset-in-set*:
x ∈# *multiset-of xs* ⟷ *x* ∈ *set xs*
⟨*proof*⟩

lemma *count-multiset-of*:
count (*multiset-of xs*) *x* = *length* (*filter* (λ*y*. *x* = *y*) *xs*)
⟨*proof*⟩

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*: $x \in set\ xs = (x :\#\ multiset\text{-}of\ xs)$
⟨*proof*⟩

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

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

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

lemma *distinct-count-atmost-1*:
$distinct\ x = (!\ a.\ count\ (multiset\text{-}of\ x)\ a = (if\ a \in set\ x\ then\ 1\ else\ 0))$
⟨*proof*⟩

lemma *multiset-of-eq-setD*:
$multiset\text{-}of\ xs = multiset\text{-}of\ ys \Longrightarrow set\ xs = set\ ys$
⟨*proof*⟩

lemma *set-eq-iff-multiset-of-eq-distinct*:
$distinct\ x \Longrightarrow distinct\ y \Longrightarrow$
$(set\ x = set\ y) = (multiset\text{-}of\ x = multiset\text{-}of\ y)$
⟨*proof*⟩

lemma *set-eq-iff-multiset-of-remdups-eq*:
$(set\ x = set\ y) = (multiset\text{-}of\ (remdups\ x) = multiset\text{-}of\ (remdups\ y))$
⟨*proof*⟩

lemma *multiset-of-compl-union* [*simp*]:
$multiset\text{-}of\ [x{\leftarrow}xs.\ P\ x] + multiset\text{-}of\ [x{\leftarrow}xs.\ \neg P\ x] = multiset\text{-}of\ xs$
⟨*proof*⟩

lemma *count-filter*:
$count\ (multiset\text{-}of\ xs)\ x = length\ [y \leftarrow xs.\ y = x]$
⟨*proof*⟩

lemma *nth-mem-multiset-of*: $i < length\ ls \Longrightarrow (ls\ !\ i) :\#\ multiset\text{-}of\ ls$
⟨*proof*⟩

lemma *multiset-of-remove1* [*simp*]:
$multiset\text{-}of\ (remove1\ a\ xs) = multiset\text{-}of\ xs - \{\#a\#\}$
⟨*proof*⟩

lemma *multiset-of-eq-length*:
assumes $multiset\text{-}of\ xs = multiset\text{-}of\ ys$
shows $length\ xs = length\ ys$
⟨*proof*⟩

lemma (**in** *linorder*) *multiset-of-insort* [*simp*]:
multiset-of (*insort x xs*) = {#*x*#} + *multiset-of xs*
⟨*proof*⟩

lemma (**in** *linorder*) *multiset-of-sort* [*simp*]:
multiset-of (*sort xs*) = *multiset-of xs*
⟨*proof*⟩

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

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

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups xs*) ≤ *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*⟩

1.5.2 Association lists – including rudimentary code generation

definition *count-of* :: (′*a* × *nat*) *list* ⇒ ′*a* ⇒ *nat* **where**
count-of xs x = (*case map-of xs x of None* ⇒ *0* | *Some n* ⇒ *n*)

lemma *count-of-multiset*:
count-of xs ∈ *multiset*
⟨*proof*⟩

lemma *count-simps* [*simp*]:
count-of [] = (λ-. *0*)
count-of ((*x*, *n*) # *xs*) = (λ*y*. *if x* = *y then n else count-of xs y*)
⟨*proof*⟩

lemma *count-of-empty*:
x ∉ *fst* ‘ *set xs* ⟹ *count-of xs x* = *0*
⟨*proof*⟩

lemma *count-of-filter*:
count-of (*filter* (*P* ∘ *fst*) *xs*) *x* = (*if P x then count-of xs x else 0*)
⟨*proof*⟩

definition *Bag* :: (*'a* × *nat*) *list* ⇒ *'a multiset* **where**
Bag xs = *Abs-multiset* (*count-of xs*)

code-datatype *Bag*

lemma *count-Bag* [*simp*, *code*]:
count (*Bag xs*) = *count-of xs*
⟨*proof*⟩

lemma *Mempty-Bag* [*code*]:
{#} = *Bag* []
⟨*proof*⟩

lemma *single-Bag* [*code*]:
{#*x*#} = *Bag* [(*x*, *1*)]
⟨*proof*⟩

lemma *MCollect-Bag* [*code*]:
MCollect (*Bag xs*) *P* = *Bag* (*filter* (*P* ∘ *fst*) *xs*)
⟨*proof*⟩

lemma *mset-less-eq-Bag* [*code*]:
Bag xs ≤ *A* ⟷ (∀(*x*, *n*) ∈ *set xs*. *count-of xs x* ≤ *count A x*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

instantiation *multiset* :: (*eq*) *eq*
begin

definition
HOL.eq A B ⟷ (*A*::*'a multiset*) ≤ *B* ∧ *B* ≤ *A*

instance ⟨*proof*⟩

end

definition (**in** *term-syntax*)
bagify :: (*'a*::*typerep* × *nat*) *list* × (*unit* ⇒ *Code-Evaluation.term*)
⇒ *'a multiset* × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *bagify xs* = *Code-Evaluation.valtermify Bag* {·} *xs*

notation *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o→* *60*)

instantiation *multiset* :: (*random*) *random*
begin

definition
Quickcheck.random i = *Quickcheck.random i o→* (λ*xs*. *Pair* (*bagify xs*))

instance ⟨*proof*⟩

end

no-notation *fcomp* (**infixl** *o>* *60*)
no-notation *scomp* (**infixl** *o→* *60*)

hide-const (**open**) *bagify*

1.6 The multiset order

1.6.1 Well-foundedness

definition *mult1* :: ('*a* × '*a*) *set* => ('*a multiset* × '*a multiset*) *set* **where**
  [*code del*]: *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**
  [*code del*]: *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*⟩

1.6.2 Closure-free presentation

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 *one-step-implies-mult-aux*:
trans r ==>
$\forall I\ J\ K.\ (size\ J = n \wedge J \neq \{\#\} \wedge (\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r))$
−−> $(I + K, I + J) \in mult\ r$
⟨*proof*⟩

lemma *one-step-implies-mult*:
trans r ==> $J \neq \{\#\}$ ==> $\forall k \in set\text{-}of\ K.\ \exists j \in set\text{-}of\ J.\ (k, j) \in r$
==> $(I + K, I + J) \in mult\ r$
⟨*proof*⟩

1.6.3 Partial-order properties

definition *less-multiset* :: *'a*::*order multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** <# *50*)
where
$M' <\# M \longleftrightarrow (M', M) \in mult\ \{(x', x).\ x' < x\}$

definition *le-multiset* :: *'a*::*order multiset* ⇒ *'a multiset* ⇒ *bool* (**infix** <=# *50*)
where
$M' <=\# M \longleftrightarrow M' <\# M \vee M' = M$

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

interpretation *multiset-order*: *order le-multiset less-multiset*
⟨*proof*⟩

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

1.6.4 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*: $B \subset\# D ==> C + B \subset\# C + (D::'a::order\ multiset)$
⟨*proof*⟩

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

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

interpretation *multiset-order*: *ordered-ab-semigroup-add plus le-multiset less-multiset*
⟨*proof*⟩

1.7 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*⟩

context *fun-left-comm*
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*:
assumes *fun-left-comm g*
shows (⋀*x y*. *h* (*g x y*) = *f x* (*h y*)) ⟹ *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A* (**is** *PROP ?P*)
⟨*proof*⟩

lemma *fold-mset-rec*:
assumes *a* ∈# *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.

1.8 Image

definition *image-mset* :: ('*a* ⇒ '*b*) ⇒ '*a multiset* ⇒ '*b multiset* **where**
image-mset f = *fold-mset* (*op* + *o single o f*) {#}

interpretation *image-left-comm*: *fun-left-comm op* + *o single o f*
⟨*proof*⟩

lemma *image-mset-empty* [*simp*]: *image-mset f* {#} = {#}
⟨*proof*⟩

lemma *image-mset-single* [*simp*]: *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*]:
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 $\{\#x + x.\ x :\# \{\#\ x :\#\ M.\ x < c\#\}\#\}$.

1.9 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in\# XS \Longrightarrow x \in\# \{\#\ y\ \#\} + XS$
and *multi-member-this*: $x \in\# \{\#\ x\ \#\} + XS$
and *multi-member-last*: $x \in\# \{\#\ x\ \#\}$
⟨*proof*⟩

definition *ms-strict* $=$ *mult pair-less*
definition [*code del*]: *ms-weak* $=$ *ms-strict* $\cup$ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
⟨*proof*⟩

lemma *smsI*:
(*set-of A*, *set-of B*) $\in$ *max-strict* $\Longrightarrow$ $(Z + A, Z + B) \in$ *ms-strict*
⟨*proof*⟩

lemma *wmsI*:

(*set-of A*, *set-of B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
⟨*proof*⟩

inductive *pw-leq*
where
pw-leq-empty: *pw-leq* {#} {#}
| *pw-leq-step*: ⟦(*x*,*y*) ∈ *pair-leq*; *pw-leq X Y* ⟧ ⟹ *pw-leq* ({#*x*#} + *X*) ({#*y*#} + *Y*)

lemma *pw-leq-lstep*:
(*x*, *y*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
⟨*proof*⟩

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-of A*, *set-of B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#}))
⟨*proof*⟩

lemma
assumes *pwleq*: *pw-leq Z Z′*
shows *ms-strictI*: (*set-of A*, *set-of B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z′* + *B*) ∈ *ms-strict*
and *ms-weakI1*: (*set-of A*, *set-of B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z′* + *B*) ∈ *ms-weak*
and *ms-weakI2*: (*Z* + {#}, *Z′* + {#}) ∈ *ms-weak*
⟨*proof*⟩

lemma *empty-idemp*: {#} + *x* = *x x* + {#} = *x*
and *nonempty-plus*: {# *x* #} + *rs* ≠ {#}
and *nonempty-single*: {# *x* #} ≠ {#}
⟨*proof*⟩

⟨*ML*⟩

1.10 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-ext-iff* [*symmetric*]

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*

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

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

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

lemma *mset-less-trans*: $(M::'a\ multiset) < K \Longrightarrow K < N \Longrightarrow M < N$
⟨*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 *mult-less-not-refl*:
$\neg\ M \subset\# (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-trans*:
$K \subset\# M \Longrightarrow M \subset\# N \Longrightarrow K \subset\# (N::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-not-sym*:
$M \subset\# N \Longrightarrow \neg\ N \subset\# (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *mult-less-asym*:
$M \subset\# N \Longrightarrow (\neg\ P \Longrightarrow N \subset\# (M::'a::order\ multiset)) \Longrightarrow P$
⟨*proof*⟩

⟨*ML*⟩

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-of* as,
multiset-of bs) $\in$ *mult* R
⟨*proof*⟩

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

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

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

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*⟩

definition *perm* :: $('a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow bool$ **where**
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

definition *fixgames* :: *ZF set* ⇒ *ZF set* **where**
fixgames A ≡ { *Opair l r* | *l r*. *explode l* ⊆ *A* & *explode r* ⊆ *A*}

definition *games-lfp* :: *ZF set* **where**
games-lfp ≡ *lfp fixgames*

definition *games-gfp* :: *ZF set* **where**
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*⟩

definition *left-option* :: *ZF* ⇒ *ZF* ⇒ *bool* **where**
left-option g opt ≡ (*Elem opt* (*Fst g*))

definition *right-option* :: *ZF* ⇒ *ZF* ⇒ *bool* **where**
right-option g opt ≡ (*Elem opt* (*Snd g*))

definition *is-option-of* :: (*ZF* ∗ *ZF*) *set* **where**
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*: $g \in games$
and *opt*: *left-option g opt* $\vee$ *right-option g opt*
shows $opt \in games$
⟨*proof*⟩

lemma *option2elem*: $(opt,g) \in$ *is-option-of* $\Longrightarrow \exists\ u\ v.$ *Elem opt u* $\wedge$ *Elem u v* $\wedge$ *Elem v g*
⟨*proof*⟩

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

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

lemma *games-gfp-imp-lfp*: $g \in$ *games-gfp* $\longrightarrow g \in$ *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*: $g \in$ *games-lfp*
and *opt*: *left-option g opt* $\vee$ *right-option g opt*
shows $opt \in$ *games-lfp*
⟨*proof*⟩

lemma *is-option-of-imp-games*:
assumes *hyp*: $(opt,\ g) \in$ *is-option-of*
shows $opt \in$ *games-lfp* $\wedge\ g \in$ *games-lfp*
⟨*proof*⟩

lemma *games-lfp-represent*: $x \in$ *games-lfp* $\Longrightarrow \exists\ l\ r.\ x =$ *Opair l r*
⟨*proof*⟩

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

definition *left-options* :: *game* $\Rightarrow$ *game zet* **where**
left-options g $\equiv$ *zimage Abs-game* (*zexplode* (*Fst* (*Rep-game g*)))

definition *right-options* :: *game* $\Rightarrow$ *game zet* **where**
right-options g $\equiv$ *zimage Abs-game* (*zexplode* (*Snd* (*Rep-game g*)))

definition *options* :: *game* $\Rightarrow$ *game zet* **where**
options g $\equiv$ *zunion* (*left-options g*) (*right-options g*)

definition *Game* :: *game zet* ⇒ *game zet* ⇒ *game* **where**
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*⟩

definition *option-of* :: (*game* ∗ *game*) *set* **where**
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*⟩

function
neg-game :: *game* ⇒ *game*
where
[*simp del*]: *neg-game g* = *Game* (*zimage neg-game* (*right-options g*)) (*zimage neg-game* (*left-options g*))
⟨*proof*⟩
termination ⟨*proof*⟩

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

function
ge-game :: (*game* ∗ *game*) ⇒ *bool*
where
[*simp del*]: *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*))
⟨*proof*⟩
termination ⟨*proof*⟩

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*⟩

definition *eq-game* :: *game* ⇒ *game* ⇒ *bool* **where**
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*: $(\bigwedge x.\ \forall y.\ (y,\ x) \in lprod\ option\text{-}of \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$
⟨*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* $\Longrightarrow$ *eq-game b c* $\Longrightarrow$ *eq-game a c*
⟨*proof*⟩

definition *zero-game* :: *game*
where *zero-game* $\equiv$ *Game zempty zempty*

function
plus-game :: *game* $\Rightarrow$ *game* $\Rightarrow$ *game*
where
[*simp del*]: *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*)))
⟨*proof*⟩
termination ⟨*proof*⟩

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

lemma *game-ext-eq*: $(G = H) = (left\text{-}options\ G = left\text{-}options\ H \wedge right\text{-}options\ G = right\text{-}options\ H)$
⟨*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*) $\Longrightarrow$ *zin opt* (*options g*)
⟨*proof*⟩

lemma *right-imp-options*[*simp*]: *zin opt* (*right-options g*) $\Longrightarrow$ *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*⟩

definition *eq-game-rel* :: (*game* ∗ *game*) *set* **where**
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* :: *ordered-ab-group-add*
⟨*proof*⟩

end