ZF

Steven Obua

October 8, 2017

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*

axiomatization where
  Empty: *Not* (*Elem x Empty*) **and**
  Ext: (*x* = *y*) = (! *z*. *Elem z x* = *Elem z y*) **and**
  Sum: *Elem z* (*Sum x*) = (*? y*. *Elem z y* & *Elem y x*) **and**
  Power: *Elem y* (*Power x*) = (*subset y x*) **and**
  Repl: *Elem b* (*Repl A f*) = (*? a*. *Elem a A* & *b* = *f a*) **and**
  Regularity: *A* ≠ *Empty* ⟶ (*? x*. *Elem x A* & (! *y*. *Elem y x* ⟶ *Not* (*Elem y*

A))) **and**
Infinity: *Elem Empty Inf* & (! x. *Elem x Inf* $\longrightarrow$ *Elem* (*SucNat x*) *Inf*)

definition *Sep* :: *ZF* $\Rightarrow$ (*ZF* $\Rightarrow$ *bool*) $\Rightarrow$ *ZF* **where**
Sep A p == (*if* (!x. *Elem x A* $\longrightarrow$ *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* $\Rightarrow$ *ZF* $\Rightarrow$ *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* $\Rightarrow$ (*ZF* $\Rightarrow$ *ZF option*) $\Rightarrow$ *ZF* **where**
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*⟩

definition *Fst* :: *ZF* $\Rightarrow$ *ZF* **where**
Fst q == *SOME* x. *?* y. q = *Opair x y*

definition *Snd* :: *ZF* $\Rightarrow$ *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*⟧ $\Longrightarrow$ ∃ *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*:∀ *x*. (∀ *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*⟩

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

primrec *nat2Nat* :: *nat* ⇒ *ZF* **where**
  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*⟩

primrec *NatInterval* :: *nat* ⇒ *nat* ⇒ *ZF* **where**
  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* ⇒ *'b* ⇒ *'a set* **where**
Ext R y ≡ { *x* . (*x*, *y*) ∈ *R* }

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

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

definition *implode* :: *ZF set* ⇒ *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* ⇒ *bool* **where**
regular R == ! *A*. *A* ≠ *Empty* ⟶ (*? x*. *Elem x A* & (! *y*. (*y*, *x*) ∈ *R* ⟶ *Not* (*Elem y A*)))

definition *set-like* :: (*ZF* ∗ *ZF*) *set* ⇒ *bool* **where**
set-like R == ! *y*. *Ext R y* ∈ *range explode*

definition *wfzf* :: (*ZF* ∗ *ZF*) *set* ⇒ *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* ⇒ *ZF*) ⇒ *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* ⇒ *ZF* ⇒ *ZF* **where**
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*⟩

primrec *Ext-ZF-n* :: (*ZF* ∗ *ZF*) *set* ⇒ *ZF* ⇒ *nat* ⇒ *ZF* **where**
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 ⟹ 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

definition *zet* = {*A* :: 'a *set* | *A f z*. *inj-on f A* ∧ *f* ‘ *A* ⊆ *explode z*}

typedef 'a *zet* = *zet* :: 'a *set set*
⟨*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*[*fundef-cong*]: ⟦ *M* = *N*; !! *x*. *zin x N* ⟹ *f x* = *g x* ⟧ ⟹ *zimage f M* = *zimage g N*
⟨*proof*⟩

end

theory *LProd*
imports *HOL−Library.Multiset*
begin

inductive-set
lprod :: ('*a* ∗ '*a*) *set* ⇒ ('*a list* ∗ '*a list*) *set*
for *R* :: ('*a* ∗ '*a*) *set*
where
lprod-single[*intro*!]: (*a*, *b*) ∈ *R* ⟹ ([*a*], [*b*]) ∈ *lprod R*
| *lprod-list*[*intro*!]: (*ah*@*at*, *bh*@*bt*) ∈ *lprod R* ⟹ (*a*,*b*) ∈ *R* ∨ *a* = *b* ⟹ (*ah*@*a*#*at*, *bh*@*b*#*bt*) ∈ *lprod R*

lemma (*as*,*bs*) ∈ *lprod R* ⟹ *length as* = *length bs*
⟨*proof*⟩

lemma (*as*, *bs*) ∈ *lprod R* ⟹ *1* ≤ *length as* ∧ *1* ≤ *length bs*
⟨*proof*⟩

lemma *lprod-subset-elem*: (*as*, *bs*) ∈ *lprod S* ⟹ *S* ⊆ *R* ⟹ (*as*, *bs*) ∈ *lprod R*
⟨*proof*⟩

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

lemma *lprod-implies-mult*: (*as*, *bs*) ∈ *lprod R* ⟹ *trans R* ⟹ (*mset as*, *mset bs*) ∈ *mult R*
⟨*proof*⟩

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

definition *gprod-2-2* :: ('*a* ∗ '*a*) *set* ⇒ (('*a* ∗ '*a*) ∗ ('*a* ∗ '*a*)) *set* **where**
  gprod-2-2 R ≡ { ((*a*,*b*), (*c*,*d*)) . (*a* = *c* ∧ (*b*,*d*) ∈ *R*) ∨ (*b* = *d* ∧ (*a*,*c*) ∈ *R*) }

definition *gprod-2-1* :: ('*a* ∗ '*a*) *set* ⇒ (('*a* ∗ '*a*) ∗ ('*a* ∗ '*a*)) *set* **where**
  gprod-2-1 R ≡ { ((*a*,*b*), (*c*,*d*)) . (*a* = *d* ∧ (*b*,*c*) ∈ *R*) ∨ (*b* = *c* ∧ (*a*,*d*) ∈ *R*) }

lemma *lprod-2-3*: (*a*, *b*) ∈ *R* ⟹ ([*a*, *c*], [*b*, *c*]) ∈ *lprod R*
  ⟨*proof*⟩

lemma *lprod-2-4*: (*a*, *b*) ∈ *R* ⟹ ([*c*, *a*], [*c*, *b*]) ∈ *lprod R*
  ⟨*proof*⟩

lemma *lprod-2-1*: (*a*, *b*) ∈ *R* ⟹ ([*c*, *a*], [*b*, *c*]) ∈ *lprod R*
  ⟨*proof*⟩

lemma *lprod-2-2*: (*a*, *b*) ∈ *R* ⟹ ([*a*, *c*], [*c*, *b*]) ∈ *lprod R*
  ⟨*proof*⟩

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

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

lemma *lprod-3-1*: **assumes** (*x*′, *x*) ∈ *R* **shows** ([*y*, *z*, *x*′], [*x*, *y*, *z*]) ∈ *lprod R*
  ⟨*proof*⟩

lemma *lprod-3-2*: **assumes** (*z*′,*z*) ∈ *R* **shows** ([*z*′, *x*, *y*], [*x*,*y*,*z*]) ∈ *lprod R*
  ⟨*proof*⟩

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

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

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

lemma *lprod-3-6*: **assumes** *y*′: (*y*′, *y*) ∈ *R* **shows** ([*x*, *z*, *y*′], [*x*, *y*, *z*]) ∈ *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**
$perm\ f\ A \equiv inj\text{-}on\ f\ A \wedge f\ `\ A = A$

lemma $((as,bs) \in lprod\ R) =$
$(\exists\ f.\ perm\ f\ \{0\ ..<\ (length\ as)\} \wedge$
$(\forall\ j.\ j < length\ as \longrightarrow ((nth\ as\ j,\ nth\ bs\ (f\ j)) \in R \vee (nth\ as\ j = nth\ bs\ (f\ j))))$
$\wedge$
$(\exists\ i.\ i < length\ as \wedge (nth\ as\ i,\ nth\ bs\ (f\ i)) \in R))$
⟨*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**
$fixgames\ A \equiv \{\ Opair\ l\ r \mid l\ r.\ explode\ l \subseteq A\ \&\ explode\ r \subseteq A\}$

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* ∈ *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*: $(opt,\ g) \in$ *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*⟩

definition *game* = *games-lfp*

typedef *game* = *game*
⟨*proof*⟩

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

definition *right-options* :: *game* ⇒ *game zet* **where**
right-options g ≡ *zimage Abs-game* (*zexplode* (*Snd* (*Rep-game g*)))

definition *options* :: *game* ⇒ *game zet* **where**
options g ≡ *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* $\Longrightarrow$ *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*: $((option,\ g) \in option\text{-}of) = ((Rep\text{-}game\ option,\ Rep\text{-}game\ g) \in is\text{-}option\text{-}of)$
⟨*proof*⟩

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

lemma *wf-option-of*[*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*) ⟹ $(x,\ g) \in option\text{-}of$
⟨*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*]:
$\forall\ y.\ (zin\ y\ (right\text{-}options\ x) \longrightarrow \neg\ ge\text{-}game\ (x,\ y)) \wedge (zin\ y\ (left\text{-}options\ x) \longrightarrow \neg\ (ge\text{-}game\ (y,\ x))) \wedge ge\text{-}game\ (x,\ x)$
⟨*proof*⟩

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

lemma $\forall\ y.\ (zin\ y\ (right\text{-}options\ x) \longrightarrow \neg\ ge\text{-}game\ (x,\ y)) \wedge (zin\ y\ (left\text{-}options\ x) \longrightarrow \neg\ (ge\text{-}game\ (y,\ x))) \wedge ge\text{-}game\ (x,\ x)$
⟨*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*: (⋀*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*⟩

definition *zero-game* :: *game*
where *zero-game* ≡ *Game zempty zempty*

function
plus-game :: *game* ⇒ *game* ⇒ *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*: $(G = H) = (\textit{left-options}\ G = \textit{left-options}\ H \wedge \textit{right-options}\ G = \textit{right-options}\ H)$
⟨*proof*⟩

lemma *left-zero-game*[*simp*]: $\textit{left-options}\ (\textit{zero-game}) = \textit{zempty}$
⟨*proof*⟩

lemma *right-zero-game*[*simp*]: $\textit{right-options}\ (\textit{zero-game}) = \textit{zempty}$
⟨*proof*⟩

lemma *plus-game-zero-right*[*simp*]: $\textit{plus-game}\ G\ \textit{zero-game} = G$
⟨*proof*⟩

lemma *plus-game-zero-left*: $\textit{plus-game}\ \textit{zero-game}\ G = G$
⟨*proof*⟩

lemma *left-imp-options*[*simp*]: $\textit{zin}\ \textit{opt}\ (\textit{left-options}\ g) \Longrightarrow \textit{zin}\ \textit{opt}\ (\textit{options}\ g)$
⟨*proof*⟩

lemma *right-imp-options*[*simp*]: $\textit{zin}\ \textit{opt}\ (\textit{right-options}\ g) \Longrightarrow \textit{zin}\ \textit{opt}\ (\textit{options}\ g)$
⟨*proof*⟩

lemma *left-options-plus*:
$\textit{left-options}\ (\textit{plus-game}\ u\ v) = \textit{zunion}\ (\textit{zimage}\ (\lambda g.\ \textit{plus-game}\ g\ v)\ (\textit{left-options}\ u))\ (\textit{zimage}\ (\lambda h.\ \textit{plus-game}\ u\ h)\ (\textit{left-options}\ v))$
⟨*proof*⟩

lemma *right-options-plus*:
$\textit{right-options}\ (\textit{plus-game}\ u\ v) = \textit{zunion}\ (\textit{zimage}\ (\lambda g.\ \textit{plus-game}\ g\ v)\ (\textit{right-options}\ u))\ (\textit{zimage}\ (\lambda h.\ \textit{plus-game}\ u\ h)\ (\textit{right-options}\ v))$
⟨*proof*⟩

lemma *left-options-neg*: $\textit{left-options}\ (\textit{neg-game}\ u) = \textit{zimage}\ \textit{neg-game}\ (\textit{right-options}\ u)$
⟨*proof*⟩

lemma *right-options-neg*: $\textit{right-options}\ (\textit{neg-game}\ u) = \textit{zimage}\ \textit{neg-game}\ (\textit{left-options}\ u)$
⟨*proof*⟩

lemma *plus-game-assoc*: $\textit{plus-game}\ (\textit{plus-game}\ F\ G)\ H = \textit{plus-game}\ F\ (\textit{plus-game}\ G\ H)$
⟨*proof*⟩

lemma *neg-plus-game*: $\textit{neg-game}\ (\textit{plus-game}\ G\ H) = \textit{plus-game}\ (\textit{neg-game}\ G)\ (\textit{neg-game}\ H)$
⟨*proof*⟩

lemma *eq-game-plus-inverse*: $\textit{eq-game}\ (\textit{plus-game}\ x\ (\textit{neg-game}\ x))\ \textit{zero-game}$

⟨*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* }

definition *Pg* = *UNIV*//*eq-game-rel*

typedef *Pg* = *Pg*
⟨*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* = *the-elem* (⋃ *g* ∈ *Rep-Pg G*. {*Abs-Pg* (*eq-game-rel* `` {*neg-game g*})})

definition
Pg-plus-def: *G* + *H* = *the-elem* (⋃ *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