ZF

Steven Obua

November 22, 2007

theory *Helper*
imports *Main*
begin

lemma *theI2′* : $?!\ x.\ P\ x \Longrightarrow (!!\ x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (THE\ x.\ P\ x)$
  apply *auto*
  apply (*subgoal-tac* P ($THE\ x.\ P\ x$))
  apply *blast*
  apply (*rule theI*)
  apply *auto*
  done

lemma *in-range-superfluous*: ($z \in$ *range* f & $z \in (f\ `\ x)) = (z \in f\ `\ x)$
  by *auto*

lemma *f-x-in-range-f*: $f\ x \in$ *range* f
  by (*blast intro*: *image-eqI*)

lemma *comp-inj*: *inj* $f \Longrightarrow$ *inj* $g \Longrightarrow$ *inj* $(g\ o\ f)$
  by (*blast intro*: *comp-inj-on subset-inj-on*)

lemma *comp-image-eq*: $(g\ o\ f)\ `\ x = g\ `\ f\ `\ x$
  by *auto*

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*: $Elem\ b\ (Sep\ A\ p) = (Elem\ b\ A\ \&\ p\ b)$
apply (*auto simp add*: *Sep-def Empty*)
apply (*auto simp add*: *Let-def Repl*)
apply (*rule someI2*, *auto*)+
done

lemma *subset-empty*: *subset Empty A*
by (*simp add*: *subset-def Empty*)

theorem *Upair*: $Elem\ x\ (Upair\ a\ b) = (x = a \mid x = b)$
apply (*auto simp add*: *Upair-def Repl*)
apply (*rule exI*[**where** *x*=*Empty*])
apply (*simp add*: *Power subset-empty*)
apply (*rule exI*[**where** *x*=*Power Empty*])

apply (*auto*)
apply (*auto simp add*: *Ext Power subset-def Empty*)
apply (*drule spec*[**where** *x*=*Empty*], *simp add*: *Empty*)+
done

lemma *Singleton*: *Elem x* (*Singleton y*) = (*x* = *y*)
by (*simp add*: *Singleton-def Upair*)

constdefs
Opair:: *ZF* ⇒ *ZF* ⇒ *ZF*
Opair a b == *Upair* (*Upair a a*) (*Upair a b*)

lemma *Upair-singleton*: (*Upair a a* = *Upair c d*) = (*a* = *c* & *a* = *d*)
by (*auto simp add*: *Ext*[**where** *x*=*Upair a a*] *Upair*)

lemma *Upair-fsteq*: (*Upair a b* = *Upair a c*) = ((*a* = *b* & *a* = *c*) | (*b* = *c*))
by (*auto simp add*: *Ext*[**where** *x*=*Upair a b*] *Upair*)

lemma *Upair-comm*: *Upair a b* = *Upair b a*
by (*auto simp add*: *Ext Upair*)

theorem *Opair*: (*Opair a b* = *Opair c d*) = (*a* = *c* & *b* = *d*)
proof −
have *fst*: (*Opair a b* = *Opair c d*) ⟹ *a* = *c*
apply (*simp add*: *Opair-def*)
apply (*simp add*: *Ext*[**where** *x*=*Upair* (*Upair a a*) (*Upair a b*)])
apply (*drule spec*[**where** *x*=*Upair a a*])
apply (*auto simp add*: *Upair Upair-singleton*)
done
show *?thesis*
apply (*auto*)
apply (*erule fst*)
apply (*frule fst*)
apply (*auto simp add*: *Opair-def Upair-fsteq*)
done
qed

constdefs
Replacement :: *ZF* ⇒ (*ZF* ⇒ *ZF option*) ⇒ *ZF*
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*)
by (*auto simp add*: *Replacement-def Repl Sep*)

constdefs
Fst :: *ZF* ⇒ *ZF*
Fst q == *SOME x*. *? y*. *q* = *Opair x y*
Snd :: *ZF* ⇒ *ZF*

Snd q == SOME y. ? x. q = Opair x y

theorem *Fst*: *Fst* (*Opair x y*) = *x*
apply (*simp add*: *Fst-def*)
apply (*rule someI2*)
apply (*simp-all add*: *Opair*)
done

theorem *Snd*: *Snd* (*Opair x y*) = *y*
apply (*simp add*: *Snd-def*)
apply (*rule someI2*)
apply (*simp-all add*: *Opair*)
done

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

lemma *isOpair*: *isOpair* (*Opair x y*) = *True*
by (*auto simp add*: *isOpair-def*)

lemma *FstSnd*: *isOpair x* ⟹ *Opair* (*Fst x*) (*Snd x*) = *x*
by (*auto simp add*: *isOpair-def Fst Snd*)

constdefs
CartProd :: *ZF* ⇒ *ZF* ⇒ *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*))
apply (*auto simp add*: *CartProd-def Sum Repl*)
apply (*rule-tac x=Repl B* (*Opair a*) **in** *exI*)
apply (*auto simp add*: *Repl*)
done

constdefs
explode :: *ZF* ⇒ *ZF set*
explode z == { *x. Elem x z* }

lemma *explode-Empty*: (*explode x* = {}) = (*x* = *Empty*)
by (*auto simp add*: *explode-def Ext Empty*)

lemma *explode-Elem*: (*x* ∈ *explode X*) = (*Elem x X*)
by (*simp add*: *explode-def*)

lemma *Elem-explode-in*: ⟦ *Elem a A*; *explode A* ⊆ *B*⟧ ⟹ *a* ∈ *B*
by (*auto simp add*: *explode-def*)

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

((*explode a*) × (*explode b*))
  by (*simp add*: *explode-def expand-set-eq CartProd image-def*)

lemma *explode-Repl-eq*: *explode* (*Repl A f*) = *image f* (*explode A*)
  by (*simp add*: *explode-def Repl image-def*)

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*)
  apply (*auto simp add*: *Domain-def Replacement*)
  apply (*rule-tac x=Snd x* **in** *exI*)
  apply (*simp add*: *FstSnd*)
  apply (*rule-tac x=Opair x y* **in** *exI*)
  apply (*simp add*: *isOpair Fst*)
  done

theorem *Range*: *Elem y* (*Range f*) = (*? x*. *Elem* (*Opair x y*) *f*)
  apply (*auto simp add*: *Range-def Replacement*)
  apply (*rule-tac x=Fst x* **in** *exI*)
  apply (*simp add*: *FstSnd*)
  apply (*rule-tac x=Opair x y* **in** *exI*)
  apply (*simp add*: *isOpair Snd*)
  done

theorem *union*: *Elem x* (*union A B*) = (*Elem x A* | *Elem x B*)
  by (*auto simp add*: *union-def Sum Upair*)

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* ⟹ (*Lambda A f*)´*x* = *f x*

by (*simp add*: *app-def Lambda-def Repl Opair*)

lemma *isFun-Lambda*: *isFun* (*Lambda A f*)
by (*auto simp add*: *isFun-def Lambda-def Repl Opair*)

lemma *domain-Lambda*: *Domain* (*Lambda A f*) = *A*
apply (*auto simp add*: *Domain-def*)
apply (*subst Ext*)
apply (*auto simp add*: *Replacement*)
apply (*simp add*: *Lambda-def Repl*)
apply (*auto simp add*: *Fst*)
apply (*simp add*: *Lambda-def Repl*)
apply (*rule-tac x*=*Opair z* (*f z*) **in** *exI*)
apply (*auto simp add*: *Fst isOpair-def*)
done

lemma *Lambda-ext*: (*Lambda s f* = *Lambda t g*) = (*s* = *t* & (! *x*. *Elem x s* ⟶ *f x* = *g x*))
proof −
have *Lambda s f* = *Lambda t g* ⟹ *s* = *t*
apply (*subst domain-Lambda*[**where** *A* = *s* **and** *f* = *f*, *symmetric*])
apply (*subst domain-Lambda*[**where** *A* = *t* **and** *f* = *g*, *symmetric*])
apply *auto*
done
then show *?thesis*
apply *auto*
apply (*subst Lambda-app*[**where** *f*=*f*, *symmetric*], *simp*)
apply (*subst Lambda-app*[**where** *f*=*g*, *symmetric*], *simp*)
apply *auto*
apply (*auto simp add*: *Lambda-def Repl Ext*)
apply (*auto simp add*: *Ext*[*symmetric*])
done
qed

constdefs
PFun :: *ZF* ⇒ *ZF* ⇒ *ZF*
PFun A B == *Sep* (*Power* (*CartProd A B*)) *isFun*
Fun :: *ZF* ⇒ *ZF* ⇒ *ZF*
Fun A B == *Sep* (*PFun A B*) (λ *f*. *Domain f* = *A*)

lemma *Fun-Range*: *Elem f* (*Fun U V*) ⟹ *subset* (*Range f*) *V*
apply (*simp add*: *Fun-def Sep PFun-def Power subset-def CartProd*)
apply (*auto simp add*: *Domain Range*)
apply (*erule-tac x*=*Opair xa x* **in** *allE*)
apply (*auto simp add*: *Opair*)
done

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

apply (*simp add*: *PFun-def Sep Power subset-def*, *clarify*)
apply (*erule-tac x=p* **in** *allE*)
apply (*auto simp add*: *CartProd isOpair Fst Snd*)
done

lemma *Fun-implies-PFun*[*simp*]: *Elem f* (*Fun U V*) $\Longrightarrow$ *Elem f* (*PFun U V*)
by (*simp add*: *Fun-def Sep*)

lemma *Elem-Elem-Fun*: *Elem F* (*Fun U V*) $\Longrightarrow$ *Elem p F* $\Longrightarrow$ *isOpair p* & *Elem* (*Fst p*) *U* & *Elem* (*Snd p*) *V*
by (*auto simp add*: *Elem-Elem-PFun dest*: *Fun-implies-PFun*)

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*
apply (*frule Elem-Elem-PFun*[**where** *p=x*], *simp*)
apply (*frule Elem-Elem-PFun*[**where** *p=y*], *simp*)
apply (*subgoal-tac isFun F*)
apply (*simp add*: *isFun-def isOpair-def*)
apply (*auto simp add*: *Fst Snd*, *blast*)
apply (*auto simp add*: *PFun-def Sep*)
done

lemma *Fun-total*: $[\![$*Elem F* (*Fun U V*); *Elem a U*$]\!]$ $\Longrightarrow$ $\exists$ *x*. *Elem* (*Opair a x*) *F*
using [[*simp-depth-limit* = *2*]]
by (*auto simp add*: *Fun-def Sep Domain*)

lemma *unique-fun-value*: $[\![$*isFun f*; *Elem x* (*Domain f*)$]\!]$ $\Longrightarrow$ *?*! *y*. *Elem* (*Opair x y*) *f*
by (*auto simp add*: *Domain isFun-def*)

lemma *fun-value-in-range*: $[\![$*isFun f*; *Elem x* (*Domain f*)$]\!]$ $\Longrightarrow$ *Elem* (*f ´x*) (*Range f*)
apply (*auto simp add*: *Range*)
apply (*drule unique-fun-value*)
apply *simp*
apply (*simp add*: *app-def*)
apply (*rule exI*[**where** *x=x*])
apply (*auto simp add*: *the-equality*)
done

lemma *fun-range-witness*: $[\![$*isFun f*; *Elem y* (*Range f*)$]\!]$ $\Longrightarrow$ *? x*. *Elem x* (*Domain f*) & *f ´x* = *y*
apply (*auto simp add*: *Range*)
apply (*rule-tac x=x* **in** *exI*)
apply (*auto simp add*: *app-def the-equality isFun-def Domain*)
done

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

apply (*rule exI*[**where** *x*= % *x*. (*THE y*. *Elem* (*Opair x y*) *F*)])
apply (*simp add*: *Ext Lambda-def Repl Domain*)
apply (*simp add*: *Ext*[*symmetric*])
apply *auto*
apply (*frule Elem-Elem-Fun*)
apply *auto*
apply (*rule-tac x*=*Fst z* **in** *exI*)
apply (*simp add*: *isOpair-def*)
apply (*auto simp add*: *Fst Snd Opair*)
apply (*rule theI2'*)
apply *auto*
apply (*drule Fun-implies-PFun*)
apply (*drule-tac x*=*Opair x ya* **and** *y*=*Opair x yb* **in** *PFun-inj*)
apply (*auto simp add*: *Fst Snd*)
apply (*drule Fun-implies-PFun*)
apply (*drule-tac x*=*Opair x y* **and** *y*=*Opair x ya* **in** *PFun-inj*)
apply (*auto simp add*: *Fst Snd*)
apply (*rule theI2'*)
apply (*auto simp add*: *Fun-total*)
apply (*drule Fun-implies-PFun*)
apply (*drule-tac x*=*Opair a x* **and** *y*=*Opair a y* **in** *PFun-inj*)
apply (*auto simp add*: *Fst Snd*)
done

lemma *Elem-Lambda-Fun*: *Elem* (*Lambda A f*) (*Fun U V*) = (*A* = *U* & (! *x*. *Elem x A* ⟶ *Elem* (*f x*) *V*))
proof −
have *Elem* (*Lambda A f*) (*Fun U V*) ⟹ *A* = *U*
by (*simp add*: *Fun-def Sep domain-Lambda*)
then show *?thesis*
apply *auto*
apply (*drule Fun-Range*)
apply (*subgoal-tac f x* = ((*Lambda U f*) ´ *x*))
prefer *2*
apply (*simp add*: *Lambda-app*)
apply *simp*
apply (*subgoal-tac Elem* (*Lambda U f* ´ *x*) (*Range* (*Lambda U f*)))
apply (*simp add*: *subset-def*)
apply (*rule fun-value-in-range*)
apply (*simp-all add*: *isFun-Lambda domain-Lambda*)
apply (*simp add*: *Fun-def Sep PFun-def Power domain-Lambda isFun-Lambda*)
apply (*auto simp add*: *subset-def CartProd*)
apply (*rule-tac x*=*Fst x* **in** *exI*)
apply (*auto simp add*: *Lambda-def Repl Fst*)
done
qed

constdefs

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

lemma cond-wf-Elem:
  assumes hyps:∀ x. (∀ y. Elem y x ⟶ Elem y U ⟶ P y) ⟶ Elem x U ⟶ P
x Elem a U
  shows P a
proof −
  {
    fix P
    fix U
    fix a
    assume P-induct: (∀ x. (∀ y. Elem y x ⟶ Elem y U ⟶ P y) ⟶ (Elem x U
⟶ P x))
    assume a-in-U: Elem a U
    have P a
      proof −
        term P
        term Sep
        let ?Z = Sep U (Not o P)
        have ?Z = Empty ⟶ P a by (simp add: Ext Sep Empty a-in-U)
        moreover have ?Z ≠ Empty ⟶ False
          proof
            assume not-empty: ?Z ≠ Empty
         note thereis-x = Regularity[where A=?Z, simplified not-empty, simplified]
           then obtain x where x-def: Elem x ?Z & (! y. Elem y x ⟶ Not (Elem
y ?Z)) ..
            then have x-induct:! y. Elem y x ⟶ Elem y U ⟶ P y by (simp add:
Sep)
            have Elem x U ⟶ P x
                 by (rule impE[OF spec[OF P-induct, where x=x], OF x-induct],
assumption)
            moreover have Elem x U & Not(P x)
              apply (insert x-def)
              apply (simp add: Sep)
              done
            ultimately show False by auto
          qed
        ultimately show P a by auto
      qed
  }
  with hyps show ?thesis by blast
qed

lemma cond2-wf-Elem:
  assumes
     special-P: ? U. ! x. Not(Elem x U) ⟶ (P x)
     and P-induct: ∀ x. (∀ y. Elem y x ⟶ P y) ⟶ P x
  shows
```

  P a
proof –
  have *? U Q. P = (λ x. (Elem x U ⟶ Q x))*
  proof –
    from *special-P* **obtain** *U* **where** *U:! x. Not(Elem x U) ⟶ (P x)* **..**
    show *?thesis*
      apply (*rule-tac exI*[**where** *x=U*])
      apply (*rule exI*[**where** *x=P*])
      apply (*rule ext*)
      apply (*auto simp add: U*)
      done
  qed
  then obtain *U* **where** *? Q. P = (λ x. (Elem x U ⟶ Q x))* **..**
  then obtain *Q* **where** *UQ: P = (λ x. (Elem x U ⟶ Q x))* **..**
  show *?thesis*
    apply (*auto simp add: UQ*)
    apply (*rule cond-wf-Elem*)
    apply (*rule P-induct*[*simplified UQ*])
    apply *simp*
    done
qed

consts
  nat2Nat :: nat ⇒ ZF

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

constdefs
  Nat2nat :: ZF ⇒ nat
  Nat2nat == inv nat2Nat

lemma *Elem-nat2Nat-inf*[*intro*]: *Elem (nat2Nat n) Inf*
  apply (*induct n*)
  apply (*simp-all add: Infinity*)
  done

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

lemma *Elem-nat2Nat-Nat*[*intro*]: *Elem (nat2Nat n) Nat*
  by (*auto simp add: Nat-def Sep*)

lemma *Elem-Empty-Nat*: *Elem Empty Nat*
  by (*auto simp add: Nat-def Sep Infinity*)

lemma *Elem-SucNat-Nat*: *Elem N Nat ⟹ Elem (SucNat N) Nat*

```
  by (auto simp add: Nat-def Sep Infinity)

lemma no-infinite-Elem-down-chain:
  Not (? f. isFun f & Domain f = Nat & (! N. Elem N Nat ⟶ Elem (f´(SucNat N)) (f´N)))
proof -
  {
    fix f
    assume f:isFun f & Domain f = Nat & (! N. Elem N Nat ⟶ Elem (f´(SucNat N)) (f´N))
    let ?r = Range f
    have ?r ≠ Empty
      apply (auto simp add: Ext Empty)
      apply (rule exI[where x=f´Empty])
      apply (rule fun-value-in-range)
      apply (auto simp add: f Elem-Empty-Nat)
      done
    then have ? x. Elem x ?r & (! y. Elem y x ⟶ Not(Elem y ?r))
      by (simp add: Regularity)
    then obtain x where x: Elem x ?r & (! y. Elem y x ⟶ Not(Elem y ?r)) ..
    then have ? N. Elem N (Domain f) & f´N = x
      apply (rule-tac fun-range-witness)
      apply (simp-all add: f)
      done
    then have ? N. Elem N Nat & f´N = x
      by (simp add: f)
    then obtain N where N: Elem N Nat & f´N = x ..
    from N have N': Elem N Nat by auto
    let ?y = f´(SucNat N)
    have Elem-y-r: Elem ?y ?r
      by (simp-all add: f Elem-SucNat-Nat N fun-value-in-range)
    have Elem ?y (f´N) by (auto simp add: f N')
    then have Elem ?y x by (simp add: N)
    with x have Not (Elem ?y ?r) by auto
    with Elem-y-r have False by auto
  }
  then show ?thesis by auto
qed

lemma Upair-nonEmpty: Upair a b ≠ Empty
  by (auto simp add: Ext Empty Upair)

lemma Singleton-nonEmpty: Singleton x ≠ Empty
  by (auto simp add: Singleton-def Upair-nonEmpty)

lemma antisym-Elem: Not(Elem a b & Elem b a)
proof -
  {
    fix a b
```

assume *ab*: *Elem a b*
assume *ba*: *Elem b a*
let *?Z* = *Upair a b*
have *?Z* ≠ *Empty* **by** (*simp add*: *Upair-nonEmpty*)
then have *? x*. *Elem x ?Z* & (! *y*. *Elem y x* ⟶ *Not*(*Elem y ?Z*))
  by (*simp add*: *Regularity*)
then obtain *x* **where** *x*:*Elem x ?Z* & (! *y*. *Elem y x* ⟶ *Not*(*Elem y ?Z*)) **..**
then have *x* = *a* ∨ *x* = *b* **by** (*simp add*: *Upair*)
moreover have *x* = *a* ⟶ *Not* (*Elem b ?Z*)
  by (*auto simp add*: *x ba*)
moreover have *x* = *b* ⟶ *Not* (*Elem a ?Z*)
  by (*auto simp add*: *x ab*)
ultimately have *False*
  by (*auto simp add*: *Upair*)
}
then show *?thesis* **by** *auto*
qed

lemma *irreflexiv-Elem*: *Not*(*Elem a a*)
  by (*simp add*: *antisym-Elem*[*of a a*, *simplified*])

lemma *antisym-Elem*: *Elem a b* ⟹ *Not* (*Elem b a*)
  apply (*insert antisym-Elem*[*of a b*])
  apply *auto*
  done

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*)
  apply (*induct m*)
  apply (*auto simp add*: *Singleton union*)
  apply (*case-tac q* <= *m*)
  apply *auto*
  apply (*subgoal-tac q* = *Suc m*)
  apply *auto*
  done

lemma *NatInterval-not-Empty*: *NatInterval n m* ≠ *Empty*
  by (*auto intro*: *n-Elem-NatInterval*[**where** *q* = *0*, *simplified*] *simp add*: *Empty Ext*)

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

apply (*case-tac ? m. n = Suc m*)
apply (*auto simp add*: *SucNat-def union Singleton*)
apply (*drule spec*[**where** *x=n − 1*])
apply *arith*
done

lemma *represent-NatInterval*[*rule-format*]: *Elem x* (*NatInterval n m*) ⟶ (*? u. n* ≤ *u* & *u* ≤ *n+m* & *nat2Nat u* = *x*)
apply (*induct m*)
apply (*auto simp add*: *Singleton union*)
apply (*rule-tac x=Suc* (*n+m*) **in** *exI*)
apply *auto*
done

lemma *inj-nat2Nat*: *inj nat2Nat*
proof −
{
fix *n m* :: *nat*
assume *nm*: *nat2Nat n* = *nat2Nat* (*n+m*)
assume *mg0*: *0* < *m*
let *?Z* = *NatInterval n m*
have *?Z* ≠ *Empty* **by** (*simp add*: *NatInterval-not-Empty*)
then have *? x*. (*Elem x ?Z*) & (! *y*. *Elem y x* ⟶ *Not* (*Elem y ?Z*))
by (*auto simp add*: *Regularity*)
then obtain *x* **where** *x*:*Elem x ?Z* & (! *y*. *Elem y x* ⟶ *Not* (*Elem y ?Z*)) **..**
then have *? u*. *n* ≤ *u* & *u* ≤ *n+m* & *nat2Nat u* = *x*
by (*simp add*: *represent-NatInterval*)
then obtain *u* **where** *u*: *n* ≤ *u* & *u* ≤ *n+m* & *nat2Nat u* = *x* **..**
have *n* < *u* ⟶ *False*
proof
assume *n-less-u*: *n* < *u*
let *?y* = *nat2Nat* (*u* − *1*)
have *Elem ?y* (*nat2Nat u*)
apply (*rule increasing-nat2Nat*)
apply (*insert n-less-u*)
apply *arith*
done
with *u* **have** *Elem ?y x* **by** *auto*
with *x* **have** *Not* (*Elem ?y ?Z*) **by** *auto*
moreover have *Elem ?y ?Z*
apply (*insert n-Elem-NatInterval*[**where** *q* = *u* − *n* − *1* **and** *n=n* **and** *m=m*])
apply (*insert n-less-u*)
apply (*insert u*)
apply *auto*
done
ultimately show *False* **by** *auto*
qed
moreover have *u* = *n* ⟶ *False*

```
    proof
      assume u = n
      with u have nat2Nat n = x by auto
      then have nm-eq-x: nat2Nat (n+m) = x by (simp add: nm)
      let ?y = nat2Nat (n+m − 1)
      have Elem ?y (nat2Nat (n+m))
        apply (rule increasing-nat2Nat)
        apply (insert mg0)
        apply arith
        done
      with nm-eq-x have Elem ?y x by auto
      with x have Not (Elem ?y ?Z) by auto
      moreover have Elem ?y ?Z
       apply (insert n-Elem-NatInterval[where q = m − 1 and n=n and m=m])
        apply (insert mg0)
        apply auto
        done
      ultimately show False by auto
    qed
    ultimately have False using u by arith
  }
  note lemma-nat2Nat = this
  have th:⋀x y. ¬ (x < y ∧ (∀(m::nat). y ≠ x + m)) by presburger
  have th′: ⋀x y. ¬ (x ≠ y ∧ (¬ x < y) ∧ (∀(m::nat). x ≠ y + m)) by presburger
  show ?thesis
    apply (auto simp add: inj-on-def)
    apply (case-tac x = y)
    apply auto
    apply (case-tac x < y)
    apply (case-tac ? m. y = x + m & 0 < m)
    apply (auto intro: lemma-nat2Nat)
    apply (case-tac y < x)
    apply (case-tac ? m. x = y + m & 0 < m)
    apply simp
    apply simp
    using th apply blast
    apply (case-tac ? m. x = y + m)
    apply (auto intro: lemma-nat2Nat)
    apply (drule sym)
    using lemma-nat2Nat apply blast
    using th′ apply blast
    done
qed

lemma Nat2nat-nat2Nat[simp]: Nat2nat (nat2Nat n) = n
  by (simp add: Nat2nat-def inv-f-f[OF inj-nat2Nat])

lemma nat2Nat-Nat2nat[simp]: Elem n Nat ⟹ nat2Nat (Nat2nat n) = n
  apply (simp add: Nat2nat-def)
```

apply (*rule-tac f-inv-f*)
apply (*auto simp add*: *image-def Nat-def Sep*)
done

lemma *Nat2nat-SucNat*: *Elem N Nat* $\Longrightarrow$ *Nat2nat* (*SucNat N*) = *Suc* (*Nat2nat N*)
apply (*auto simp add*: *Nat-def Sep Nat2nat-def*)
apply (*auto simp add*: *inv-f-f*[*OF inj-nat2Nat*])
apply (*simp only*: *nat2Nat.simps*[*symmetric*])
apply (*simp only*: *inv-f-f*[*OF inj-nat2Nat*])
done

lemma *Elem-Opair-exists*: *? z. Elem x z* & *Elem y z* & *Elem z* (*Opair x y*)
apply (*rule exI*[**where** *x*=*Upair x y*])
by (*simp add*: *Upair Opair-def*)

lemma *UNIV-is-not-in-ZF*: *UNIV* $\neq$ *explode R*
proof
let *?Russell* = { *x. Not*(*Elem x x*) }
have *?Russell* = *UNIV* **by** (*simp add*: *irreflexiv-Elem*)
moreover assume *UNIV* = *explode R*
ultimately have *russell*: *?Russell* = *explode R* **by** *simp*
then show *False*
proof(*cases Elem R R*)
case *True*
then show *?thesis*
by (*insert irreflexiv-Elem*, *auto*)
next
case *False*
then have *R* $\in$ *?Russell* **by** *auto*
then have *Elem R R* **by** (*simp add*: *russell explode-def*)
with *False* **show** *?thesis* **by** *auto*
qed
qed

constdefs
SpecialR :: (*ZF* $*$ *ZF*) *set*
SpecialR $\equiv$ { (*x*, *y*) . *x* $\neq$ *Empty* $\wedge$ *y* = *Empty*}

lemma *wf SpecialR*
apply (*subst wf-def*)
apply (*auto simp add*: *SpecialR-def*)
done

constdefs
Ext :: (*'a* $*$ *'b*) *set* $\Rightarrow$ *'b* $\Rightarrow$ *'a set*

Ext R y ≡ { *x* . (*x*, *y*) ∈ *R* }

lemma *Ext-Elem*: *Ext is-Elem-of* = *explode*
by (*auto intro*: *ext simp add*: *Ext-def is-Elem-of-def explode-def*)

lemma *Ext SpecialR Empty* ≠ *explode z*
proof
have *Ext SpecialR Empty* = *UNIV* − {*Empty*}
by (*auto simp add*: *Ext-def SpecialR-def*)
moreover assume *Ext SpecialR Empty* = *explode z*
ultimately have *UNIV* = *explode*(*union z* (*Singleton Empty*))
by (*auto simp add*: *explode-def union Singleton*)
then show *False* **by** (*simp add*: *UNIV-is-not-in-ZF*)
qed

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

lemma *inj-explode*: *inj explode*
by (*auto simp add*: *inj-on-def explode-def Ext*)

lemma *implode-explode*[*simp*]: *implode* (*explode x*) = *x*
by (*simp add*: *implode-def inj-explode*)

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*
by (*simp add*: *regular-def is-Elem-of-def Regularity*)

lemma *set-like-Elem*: *set-like is-Elem-of*
by (*auto simp add*: *set-like-def image-def Ext-Elem*)

lemma *wfzf-is-Elem-of*: *wfzf is-Elem-of*
by (*auto simp add*: *wfzf-def regular-Elem set-like-Elem*)

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

lemma *SeqSum*: *Elem x* (*SeqSum f*) = (? *n*. *Elem x* (*f n*))
apply (*auto simp add*: *SeqSum-def Sum Repl*)

```
  apply (rule-tac x = f n in exI)
  apply auto
  done

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)
  apply (auto)
  apply (simp-all add: explode-def)
  done

lemma Elem-Ext-ZF: set-like R ⟹ Elem x (Ext-ZF R s) = ((x,s) ∈ R)
  apply (simp add: Ext-ZF-def)
  apply (subst Elem-implode)
  apply (simp add: set-like-def)
  apply (simp add: Ext-def)
  done

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))
  by (simp add: Ext-ZF-hull-def SeqSum)

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 −
  from Elem-Ext-ZF-hull[OF set-like-R] x-hull
  have ? n. Elem x (Ext-ZF-n R S n) by auto
  then obtain n where n:Elem x (Ext-ZF-n R S n) ..
  with y-R-x have Elem y (Ext-ZF-n R S (Suc n))
    apply (auto simp add: Repl Sum)
    apply (rule-tac x=Ext-ZF R x in exI)
    apply (auto simp add: Elem-Ext-ZF[OF set-like-R])
```

```
    done
  with Elem-Ext-ZF-hull[OF set-like-R, where x=y] show ?thesis
    by (auto simp del: Ext-ZF-n.simps)
qed

lemma wfzf-minimal:
  assumes hyps: wfzf R C ≠ {}
  shows ∃ x. x ∈ C ∧ (∀ y. (y, x) ∈ R ⟶ y ∉ C)
proof −
  from hyps have ∃ S. S ∈ C by auto
  then obtain S where S:S ∈ C by auto
  let ?T = Sep (Ext-ZF-hull R S) (λ s. s ∈ C)
  from hyps have set-like-R: set-like R by (simp add: wfzf-def)
  show ?thesis
  proof (cases ?T = Empty)
    case True
    then have ∀ z. ¬ (Elem z (Sep (Ext-ZF R S) (λ s. s ∈ C)))
      apply (auto simp add: Ext Empty Sep Ext-ZF-hull-def SeqSum)
      apply (erule-tac x=z in allE, auto)
      apply (erule-tac x=0 in allE, auto)
      done
    then show ?thesis
      apply (rule-tac exI[where x=S])
      apply (auto simp add: Sep Empty S)
      apply (erule-tac x=y in allE)
      apply (simp add: set-like-R Elem-Ext-ZF)
      done
  next
    case False
    from hyps have regular-R: regular R by (simp add: wfzf-def)
    from
      regular-R[simplified regular-def, rule-format, OF False, simplified Sep]
      Elem-Elem-Ext-ZF-hull[OF set-like-R]
    show ?thesis by blast
  qed
qed

lemma wfzf-implies-wf: wfzf R ⟹ wf R
proof (subst wf-def, rule allI)
  assume wfzf: wfzf R
  fix P :: ZF ⇒ bool
  let ?C = {x. P x}
  {
    assume induct: (∀ x. (∀ y. (y, x) ∈ R ⟶ P y) ⟶ P x)
    let ?C = {x. ¬ (P x)}
    have ?C = {}
    proof (rule ccontr)
      assume C: ?C ≠ {}
      from
```

wfzf-minimal[*OF wfzf C*]
obtain x **where** x: $x \in$ *?C* $\wedge$ ($\forall y$. (y, x) $\in R \longrightarrow y \notin$ *?C*) **..**
then have $P\ x$
apply (*rule-tac induct*[*rule-format*])
apply *auto*
done
with x **show** *False* **by** *auto*
qed
then have ! x. $P\ x$ **by** *auto*
}
then show ($\forall x$. ($\forall y$. (y, x) $\in R \longrightarrow P\ y$) $\longrightarrow P\ x$) $\longrightarrow$ (! x. $P\ x$) **by** *blast*
qed

lemma *wf-is-Elem-of*: *wf is-Elem-of*
by (*auto simp add*: *wfzf-is-Elem-of wfzf-implies-wf*)

lemma *in-Ext-RTrans-implies-Elem-Ext-ZF-hull*:
set-like R $\Longrightarrow x \in$ (*Ext* (R ^+) s) $\Longrightarrow$ *Elem x* (*Ext-ZF-hull R s*)
apply (*simp add*: *Ext-def Elem-Ext-ZF-hull*)
apply (*erule converse-trancl-induct*[**where** *r=R*])
apply (*rule exI*[**where** *x=0*])
apply (*simp add*: *Elem-Ext-ZF*)
apply *auto*
apply (*rule-tac x=Suc n* **in** *exI*)
apply (*simp add*: *Sum Repl*)
apply (*rule-tac x=Ext-ZF R z* **in** *exI*)
apply (*auto simp add*: *Elem-Ext-ZF*)
done

lemma *implodeable-Ext-trancl*: *set-like R* $\Longrightarrow$ *set-like* (R ^+)
apply (*subst set-like-def*)
apply (*auto simp add*: *image-def*)
apply (*rule-tac x=Sep* (*Ext-ZF-hull R y*) (λ z. $z \in$ (*Ext* (R ^+) y)) **in** *exI*)
apply (*auto simp add*: *explode-def Sep set-ext*
in-Ext-RTrans-implies-Elem-Ext-ZF-hull)
done

lemma *Elem-Ext-ZF-hull-implies-in-Ext-RTrans*[*rule-format*]:
set-like R $\Longrightarrow$! x. *Elem x* (*Ext-ZF-n R s n*) $\longrightarrow x \in$ (*Ext* (R ^+) s)
apply (*induct-tac n*)
apply (*auto simp add*: *Elem-Ext-ZF Ext-def Sum Repl*)
done

lemma *set-like R* $\Longrightarrow$ *Ext-ZF* (R ^+) s = *Ext-ZF-hull R s*
apply (*frule implodeable-Ext-trancl*)
apply (*auto simp add*: *Ext*)
apply (*erule in-Ext-RTrans-implies-Elem-Ext-ZF-hull*)
apply (*simp add*: *Elem-Ext-ZF Ext-def*)
apply (*auto simp add*: *Elem-Ext-ZF Elem-Ext-ZF-hull*)

apply (*erule Elem-Ext-ZF-hull-implies-in-Ext-RTrans*[*simplified Ext-def*, *simplified*], *assumption*)
done

lemma *wf-implies-regular*: *wf R* $\Longrightarrow$ *regular R*
proof (*simp add*: *regular-def*, *rule allI*)
assume *wf*: *wf R*
fix *A*
show $A \neq Empty \longrightarrow (\exists x.\ Elem\ x\ A \wedge (\forall y.\ (y, x) \in R \longrightarrow \neg\ Elem\ y\ A))$
proof
assume *A*: $A \neq Empty$
then have $?\ x.\ x \in explode\ A$
by (*auto simp add*: *explode-def Ext Empty*)
then obtain *x* **where** $x{:}x \in explode\ A$ **..**
from *iffD1*[*OF wf-eq-minimal wf*, *rule-format*, **where** *Q*=*explode A*, *OF x*]
obtain *z* **where** $z \in explode\ A \wedge (\forall y.\ (y, z) \in R \longrightarrow y \notin explode\ A)$ **by** *auto*

then show $\exists x.\ Elem\ x\ A \wedge (\forall y.\ (y, x) \in R \longrightarrow \neg\ Elem\ y\ A)$
apply (*rule-tac exI*[**where** $x = z$])
apply (*simp add*: *explode-def*)
done
qed
qed

lemma *wf-eq-wfzf*: $(wf\ R \wedge set\text{-}like\ R) = wfzf\ R$
apply (*auto simp add*: *wfzf-implies-wf*)
apply (*auto simp add*: *wfzf-def wf-implies-regular*)
done

lemma *wfzf-trancl*: *wfzf R* $\Longrightarrow$ *wfzf* ($R\hat{\ }{+}$)
by (*auto simp add*: *wf-eq-wfzf*[*symmetric*] *implodeable-Ext-trancl wf-trancl*)

lemma *Ext-subset-mono*: $R \subseteq S \Longrightarrow Ext\ R\ y \subseteq Ext\ S\ y$
by (*auto simp add*: *Ext-def*)

lemma *set-like-subset*: *set-like R* $\Longrightarrow$ $S \subseteq R$ $\Longrightarrow$ *set-like S*
apply (*auto simp add*: *set-like-def*)
apply (*erule-tac x*=*y* **in** *allE*)
apply (*drule-tac y*=*y* **in** *Ext-subset-mono*)
apply (*auto simp add*: *image-def*)
apply (*rule-tac x*=*Sep x* (% *z*. $z \in (Ext\ S\ y)$) **in** *exI*)
apply (*auto simp add*: *explode-def Sep*)
done

lemma *wfzf-subset*: *wfzf S* $\Longrightarrow$ $R \subseteq S$ $\Longrightarrow$ *wfzf R*
by (*auto intro*: *set-like-subset wf-subset simp add*: *wf-eq-wfzf*[*symmetric*])

end

theory *Zet*
imports *HOLZF*
begin

typedef $'a\ zet = \{A :: 'a\ set \mid A\ f\ z.\ inj\text{-}on\ f\ A \wedge f\ `\ A \subseteq explode\ z\}$
  by *blast*

constdefs
  $zin :: 'a \Rightarrow 'a\ zet \Rightarrow bool$
  $zin\ x\ A == x \in (Rep\text{-}zet\ A)$

lemma *zet-ext-eq*: $(A = B) = (!\ x.\ zin\ x\ A = zin\ x\ B)$
  by (*auto simp add*: *Rep-zet-inject*[*symmetric*] *zin-def*)

constdefs
  $zimage :: ('a \Rightarrow 'b) \Rightarrow 'a\ zet \Rightarrow 'b\ zet$
  $zimage\ f\ A == Abs\text{-}zet\ (image\ f\ (Rep\text{-}zet\ A))$

lemma *zet-def'*: $zet = \{A :: 'a\ set \mid A\ f\ z.\ inj\text{-}on\ f\ A \wedge f\ `\ A = explode\ z\}$
  apply (*rule set-ext*)
  apply (*auto simp add*: *zet-def*)
  apply (*rule-tac x=f* **in** *exI*)
  apply *auto*
  apply (*rule-tac* $x{=}Sep\ z\ (\lambda\ y.\ y \in (f\ `\ x))$ **in** *exI*)
  apply (*auto simp add*: *explode-def Sep*)
  done

lemma *image-Inv-f-f*: $inj\text{-}on\ f\ B \Longrightarrow A \subseteq B \Longrightarrow (Inv\ B\ f)\ `\ f\ `\ A = A$
  apply (*rule set-ext*)
  apply (*auto simp add*: *Inv-f-f image-def*)
  apply (*rule-tac x=f x* **in** *exI*)
  apply (*auto simp add*: *Inv-f-f*)
  done

lemma *image-zet-rep*: $A \in zet \Longrightarrow ?\ z\ .\ g\ `\ A = explode\ z$
  apply (*auto simp add*: *zet-def'*)
  apply (*rule-tac* $x{=}Repl\ z\ (g\ o\ (Inv\ A\ f))$ **in** *exI*)
  apply (*simp add*: *explode-Repl-eq*)
  apply (*subgoal-tac* $explode\ z = f\ `\ A$)
  apply (*simp-all add*: *comp-image-eq image-Inv-f-f*)
  done

lemma *Inv-f-f-mem*:
  assumes $x \in A$
  shows $Inv\ A\ g\ (g\ x) \in A$
  apply (*simp add*: *Inv-def*)
  apply (*rule someI2*)

```
  using ‹x ∈ A› apply auto
  done

lemma zet-image-mem:
  assumes Azet: A ∈ zet
  shows g ‘ A ∈ zet
proof −
  from Azet have ? (f :: - ⇒ ZF). inj-on f A
    by (auto simp add: zet-def')
  then obtain f where injf: inj-on (f :: - ⇒ ZF) A
    by auto
  let ?w = f o (Inv A g)
  have subset: (Inv A g) ‘ (g ‘ A) ⊆ A
    by (auto simp add: Inv-f-f-mem)
  have inj-on (Inv A g) (g ‘ A) by (simp add: inj-on-Inv)
  then have injw: inj-on ?w (g ‘ A)
    apply (rule comp-inj-on)
    apply (rule subset-inj-on[where B=A])
    apply (auto simp add: subset injf)
    done
  show ?thesis
    apply (simp add: zet-def' comp-image-eq[symmetric])
    apply (rule exI[where x=?w])
    apply (simp add: injw image-zet-rep Azet)
    done
qed

lemma Rep-zimage-eq: Rep-zet (zimage f A) = image f (Rep-zet A)
  apply (simp add: zimage-def)
  apply (subst Abs-zet-inverse)
  apply (simp-all add: Rep-zet zet-image-mem)
  done

lemma zimage-iff: zin y (zimage f A) = (? x. zin x A & y = f x)
  by (auto simp add: zin-def Rep-zimage-eq)

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
  by (rule image-zet-rep[where g=λ x. x,OF Rep-zet, simplified])

lemma zexplode-zimplode: zexplode (zimplode A) = A
  apply (simp add: zimplode-def zexplode-def)
  apply (simp add: implode-def)
  apply (subst f-inv-f[where y=Rep-zet A])
```

apply (*auto simp add*: *Rep-zet-inverse Rep-zet-eq-explode image-def*)
done

lemma *explode-mem-zet*: *explode z* $\in$ *zet*
apply (*simp add*: *zet-def'*)
apply (*rule-tac x*=% *x*. *x* **in** *exI*)
apply (*auto simp add*: *inj-on-def*)
done

lemma *zimplode-zexplode*: *zimplode* (*zexplode z*) = *z*
apply (*simp add*: *zimplode-def zexplode-def*)
apply (*subst Abs-zet-inverse*)
apply (*auto simp add*: *explode-mem-zet implode-explode*)
done

lemma *zin-zexplode-eq*: *zin x* (*zexplode A*) = *Elem x A*
apply (*simp add*: *zin-def zexplode-def*)
apply (*subst Abs-zet-inverse*)
apply (*simp-all add*: *explode-Elem explode-mem-zet*)
done

lemma *comp-zimage-eq*: *zimage g* (*zimage f A*) = *zimage* (*g o f*) *A*
apply (*simp add*: *zimage-def*)
apply (*subst Abs-zet-inverse*)
apply (*simp-all add*: *comp-image-eq zet-image-mem Rep-zet*)
done

constdefs
zunion :: '*a zet* $\Rightarrow$ '*a zet* $\Rightarrow$ '*a zet*
zunion a b $\equiv$ *Abs-zet* ((*Rep-zet a*) $\cup$ (*Rep-zet b*))
zsubset :: '*a zet* $\Rightarrow$ '*a zet* $\Rightarrow$ *bool*
zsubset a b $\equiv$! *x*. *zin x a* $\longrightarrow$ *zin x b*

lemma *explode-union*: *explode* (*union a b*) = (*explode a*) $\cup$ (*explode b*)
apply (*rule set-ext*)
apply (*simp add*: *explode-def union*)
done

lemma *Rep-zet-zunion*: *Rep-zet* (*zunion a b*) = (*Rep-zet a*) $\cup$ (*Rep-zet b*)
proof −
from *Rep-zet*[*of a*] **have** *? f z*. *inj-on f* (*Rep-zet a*) $\wedge$ *f* ‘ (*Rep-zet a*) = *explode z*
by (*auto simp add*: *zet-def'*)
then obtain *fa za* **where** *a*:*inj-on fa* (*Rep-zet a*) $\wedge$ *fa* ‘ (*Rep-zet a*) = *explode za*
by *blast*
from *a* **have** *fa*: *inj-on fa* (*Rep-zet a*) **by** *blast*
from *a* **have** *za*: *fa* ‘ (*Rep-zet a*) = *explode za* **by** *blast*
from *Rep-zet*[*of b*] **have** *? f z*. *inj-on f* (*Rep-zet b*) $\wedge$ *f* ‘ (*Rep-zet b*) = *explode z*
by (*auto simp add*: *zet-def'*)

then obtain *fb zb* **where** *b*:*inj-on fb* (*Rep-zet b*) ∧ *fb* ' (*Rep-zet b*) = *explode zb*
by *blast*
from *b* **have** *fb*: *inj-on fb* (*Rep-zet b*) **by** *blast*
from *b* **have** *zb*: *fb* ' (*Rep-zet b*) = *explode zb* **by** *blast*
let *?f* = (λ *x*. *if x* ∈ (*Rep-zet a*) *then Opair* (*fa x*) (*Empty*) *else Opair* (*fb x*) (*Singleton Empty*))
let *?z* = *CartProd* (*union za zb*) (*Upair Empty* (*Singleton Empty*))
have *se*: *Singleton Empty* ≠ *Empty*
apply (*auto simp add*: *Ext Singleton*)
apply (*rule exI*[**where** *x*=*Empty*])
apply (*simp add*: *Empty*)
done
show *?thesis*
apply (*simp add*: *zunion-def*)
apply (*subst Abs-zet-inverse*)
apply (*auto simp add*: *zet-def*)
apply (*rule exI*[**where** *x* = *?f*])
apply (*rule conjI*)
apply (*auto simp add*: *inj-on-def Opair inj-onD*[*OF fa*] *inj-onD*[*OF fb*] *se se*[*symmetric*])
apply (*rule exI*[**where** *x* = *?z*])
apply (*insert za zb*)
apply (*auto simp add*: *explode-def CartProd union Upair Opair*)
done
qed

lemma *zunion*: *zin x* (*zunion a b*) = ((*zin x a*) ∨ (*zin x b*))
by (*auto simp add*: *zin-def Rep-zet-zunion*)

lemma *zimage-zexplode-eq*: *zimage f* (*zexplode z*) = *zexplode* (*Repl z f*)
by (*simp add*: *zet-ext-eq zin-zexplode-eq Repl zimage-iff*)

lemma *range-explode-eq-zet*: *range explode* = *zet*
apply (*rule set-ext*)
apply (*auto simp add*: *explode-mem-zet*)
apply (*drule image-zet-rep*)
apply (*simp add*: *image-def*)
apply *auto*
apply (*rule-tac x*=*z* **in** *exI*)
apply *auto*
done

lemma *Elem-zimplode*: (*Elem x* (*zimplode z*)) = (*zin x z*)
apply (*simp add*: *zimplode-def*)
apply (*subst Elem-implode*)
apply (*simp-all add*: *zin-def Rep-zet range-explode-eq-zet*)
done

constdefs

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

lemma *zempty*[*simp*]: ¬ (*zin x zempty*)
by (*auto simp add*: *zin-def zempty-def Abs-zet-inverse zet-def*)

lemma *zimage-zempty*[*simp*]: *zimage f zempty* = *zempty*
by (*auto simp add*: *zet-ext-eq zimage-iff*)

lemma *zunion-zempty-left*[*simp*]: *zunion zempty a* = *a*
by (*simp add*: *zet-ext-eq zunion*)

lemma *zunion-zempty-right*[*simp*]: *zunion a zempty* = *a*
by (*simp add*: *zet-ext-eq zunion*)

lemma *zimage-id*[*simp*]: *zimage id A* = *A*
by (*simp add*: *zet-ext-eq zimage-iff*)

lemma *zimage-cong*[*recdef-cong*]: ⟦ *M* = *N*; !! *x*. *zin x N* ⟹ *f x* = *g x* ⟧ ⟹ *zimage f M* = *zimage g N*
by (*auto simp add*: *zet-ext-eq zimage-iff*)

end

1 Multisets

theory *Multiset*
imports *Main*
begin

1.1 The type of multisets

typedef *'a multiset* = {*f*::*'a* => *nat*. *finite* {*x* . *f x* > *0*}}
proof
show (λ*x*. *0*::*nat*) ∈ *?multiset* **by** *simp*
qed

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**
{#*a*#} = *Abs-multiset* (λ*b*. *if b* = *a then 1 else 0*)

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* == *count M a* > *0*

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

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

instance *multiset* :: (*type*) {*plus*, *minus*, *zero*, *size*}
union-def: *M* + *N* == *Abs-multiset* (λa. *Rep-multiset M a* + *Rep-multiset N a*)
diff-def: *M* − *N* == *Abs-multiset* (λa. *Rep-multiset M a* − *Rep-multiset N a*)
Zero-multiset-def [*simp*]: *0* == {#}
size-def: *size M* == *setsum* (*count M*) (*set-of M*) **..**

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

Preservation of the representing set *multiset*.

lemma *const0-in-multiset* [*simp*]: (λa. *0*) ∈ *multiset*
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset* [*simp*]: (λb. *if b* = *a then 1 else 0*) ∈ *multiset*
by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> *N* ∈ *multiset* ==> (λa. *M a* + *N a*) ∈ *multiset*
apply (*simp add*: *multiset-def*)
apply (*drule* (*1*) *finite-UnI*)
apply (*simp del*: *finite-Un add*: *Un-def*)
done

lemma *diff-preserves-multiset* [*simp*]:
M ∈ *multiset* ==> (λa. *M a* − *N a*) ∈ *multiset*
apply (*simp add*: *multiset-def*)

apply (*rule finite-subset*)
 apply *auto*
done

1.2 Algebraic properties of multisets

1.2.1 Union

lemma *union-empty* [*simp*]: $M + \{\#\} = M \wedge \{\#\} + M = M$
 by (*simp add*: *union-def Mempty-def*)

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
 by (*simp add*: *union-def add-ac*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
 by (*simp add*: *union-def add-ac*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
proof −
 have $M + (N + K) = (N + K) + M$
  by (*rule union-commute*)
 also have $\ldots = N + (K + M)$
  by (*rule union-assoc*)
 also have $K + M = M + K$
  by (*rule union-commute*)
 finally show *?thesis* **.**
qed

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

instance *multiset* :: (*type*) *comm-monoid-add*
proof
 fix $a\ b\ c$ :: *'a multiset*
 show $(a + b) + c = a + (b + c)$ **by** (*rule union-assoc*)
 show $a + b = b + a$ **by** (*rule union-commute*)
 show $0 + a = a$ **by** *simp*
qed

1.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
 by (*simp add*: *Mempty-def diff-def*)

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
 by (*simp add*: *union-def diff-def*)

1.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* $\{\#\}\ a = 0$
 by (*simp add*: *count-def Mempty-def*)

lemma *count-single* [*simp*]: *count* {#*b*#} *a* = (*if b* = *a then 1 else 0*)
by (*simp add*: *count-def single-def*)

lemma *count-union* [*simp*]: *count* (*M* + *N*) *a* = *count M a* + *count N a*
by (*simp add*: *count-def union-def*)

lemma *count-diff* [*simp*]: *count* (*M* − *N*) *a* = *count M a* − *count N a*
by (*simp add*: *count-def diff-def*)

1.2.4 Set of elements

lemma *set-of-empty* [*simp*]: *set-of* {#} = {}
by (*simp add*: *set-of-def*)

lemma *set-of-single* [*simp*]: *set-of* {#*b*#} = {*b*}
by (*simp add*: *set-of-def*)

lemma *set-of-union* [*simp*]: *set-of* (*M* + *N*) = *set-of M* ∪ *set-of N*
by (*auto simp add*: *set-of-def*)

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of M* = {}) = (*M* = {#})
by (*auto simp add*: *set-of-def Mempty-def count-def expand-fun-eq*)

lemma *mem-set-of-iff* [*simp*]: (*x* ∈ *set-of M*) = (*x* :# *M*)
by (*auto simp add*: *set-of-def*)

1.2.5 Size

lemma *size-empty* [*simp*]: *size* {#} = *0*
by (*simp add*: *size-def*)

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
by (*simp add*: *size-def*)

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
using *Rep-multiset* [*of M*]
by (*simp add*: *multiset-def set-of-def count-def*)

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count N*) (*A* ∩ *set-of N*) = *setsum* (*count N*) *A*
apply (*induct rule*: *finite-induct*)
apply *simp*
apply (*simp add*: *Int-insert-left set-of-def*)
done

lemma *size-union* [*simp*]: *size* (*M* + *N*::′*a multiset*) = *size M* + *size N*
apply (*unfold size-def*)
apply (*subgoal-tac count* (*M* + *N*) = (λ*a*. *count M a* + *count N a*))
prefer *2*

apply (*rule ext*, *simp*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-Un-nat setsum-addf setsum-count-Int*)
apply (*subst Int-commute*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-count-Int*)
done

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
apply (*unfold size-def Mempty-def count-def*, *auto*)
apply (*simp add*: *set-of-def count-def expand-fun-eq*)
done

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ==> ∃ *a*. *a* :# *M*
apply (*unfold size-def*)
apply (*drule setsum-SucD*, *auto*)
done

1.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: (*M* = *N*) = (∀ *a*. *count M a* = *count N a*)
by (*simp add*: *count-def expand-fun-eq*)

lemma *single-not-empty* [*simp*]: {#*a*#} ≠ {#} ∧ {#} ≠ {#*a*#}
by (*simp add*: *single-def Mempty-def expand-fun-eq*)

lemma *single-eq-single* [*simp*]: ({#*a*#} = {#*b*#}) = (*a* = *b*)
by (*auto simp add*: *single-def expand-fun-eq*)

lemma *union-eq-empty* [*iff*]: (*M* + *N* = {#}) = (*M* = {#} ∧ *N* = {#})
by (*auto simp add*: *union-def Mempty-def expand-fun-eq*)

lemma *empty-eq-union* [*iff*]: ({#} = *M* + *N*) = (*M* = {#} ∧ *N* = {#})
by (*auto simp add*: *union-def Mempty-def expand-fun-eq*)

lemma *union-right-cancel* [*simp*]: (*M* + *K* = *N* + *K*) = (*M* = (*N*::′*a multiset*))
by (*simp add*: *union-def expand-fun-eq*)

lemma *union-left-cancel* [*simp*]: (*K* + *M* = *K* + *N*) = (*M* = (*N*::′*a multiset*))
by (*simp add*: *union-def expand-fun-eq*)

lemma *union-is-single*:
(*M* + *N* = {#*a*#}) = (*M* = {#*a*#} ∧ *N*={#} ∨ *M* = {#} ∧ *N* = {#*a*#})
apply (*simp add*: *Mempty-def single-def union-def add-is-1 expand-fun-eq*)
apply *blast*
done

lemma *single-is-union*:
({#*a*#} = *M* + *N*) = ({#*a*#} = *M* ∧ *N* = {#} ∨ *M* = {#} ∧ {#*a*#} = *N*)
apply (*unfold Mempty-def single-def union-def*)

apply (*simp add*: *add-is-1 one-is-add expand-fun-eq*)
apply (*blast dest*: *sym*)
done

lemma *add-eq-conv-diff*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\})$
using [[*simproc del*: *neq*]]
apply (*unfold single-def union-def diff-def*)
apply (*simp* (*no-asm*) *add*: *expand-fun-eq*)
apply (*rule conjI*, *force*, *safe*, *simp-all*)
apply (*simp add*: *eq-sym-conv*)
done

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

instance *multiset* :: (*type*) *cancel-ab-semigroup-add*
proof
fix $a\ b\ c$ :: $'a$ *multiset*
show $a + b = a + c \Longrightarrow b = c$ **by** *simp*
qed

1.2.7 Intersection

lemma *multiset-inter-count*:
count $(A \#\cap B)\ x = min$ (*count* $A\ x$) (*count* $B\ x$)
by (*simp add*: *multiset-inter-def min-def*)

lemma *multiset-inter-commute*: $A \#\cap B = B \#\cap A$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-max.inf-commute*)

lemma *multiset-inter-assoc*: $A \#\cap (B \#\cap C) = A \#\cap B \#\cap C$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-max.inf-assoc*)

lemma *multiset-inter-left-commute*: $A \#\cap (B \#\cap C) = B \#\cap (A \#\cap C)$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*)

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

lemma *multiset-union-diff-commute*: $B \#\cap C = \{\#\} \Longrightarrow A + B - C = A - C + B$
apply (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def split*: *split-if-asm*)

```
  apply clarsimp
  apply (erule-tac x = a in allE)
  apply auto
  done
```

1.3 Induction over multisets

```
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)
  apply (induct rule: finite-induct)
   apply auto
  apply (drule-tac a = a in mk-disjoint-insert, auto)
  done

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
  apply (unfold multiset-def)
  apply (induct-tac n, simp, clarify)
   apply (subgoal-tac f = (λa.0))
    apply simp
    apply (rule 1)
   apply (rule ext, force, clarify)
  apply (frule setsum-SucD, clarify)
  apply (rename-tac a)
  apply (subgoal-tac finite {x. (f (a := f a − 1)) x > 0})
   prefer 2
   apply (rule finite-subset)
    prefer 2
    apply assumption
   apply simp
   apply blast
  apply (subgoal-tac f = (f (a := f a − 1))(a := (f (a := f a − 1)) a + 1))
   prefer 2
   apply (rule ext)
   apply (simp (no-asm-simp))
   apply (erule ssubst, rule 2 [unfolded multiset-def], blast)
  apply (erule allE, erule impE, erule-tac [2] mp, blast)
  apply (simp (no-asm-simp) add: setsum-decr del: fun-upd-apply One-nat-def)
  apply (subgoal-tac {x. x ≠ a −−> f x ≠ 0} = {x. f x ≠ 0})
   prefer 2
   apply blast
  apply (subgoal-tac {x. x ≠ a ∧ f x ≠ 0} = {x. f x ≠ 0} − {a})
   prefer 2
   apply blast
  apply (simp add: le-imp-diff-is-add setsum-diff1-nat cong: conj-cong)
  done
```

theorem *rep-multiset-induct*:
  $f \in$ *multiset* ==> P ($\lambda a.$ *0*) ==>
    (!!f b. $f \in$ *multiset* ==> P f ==> P (f (b := f b + *1*))) ==> P f
  using *rep-multiset-induct-aux* **by** *blast*

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 −
  note *defns* = *union-def single-def Mempty-def*
  show *?thesis*
    apply (*rule Rep-multiset-inverse* [*THEN subst*])
    apply (*rule Rep-multiset* [*THEN rep-multiset-induct*])
     apply (*rule empty* [*unfolded defns*])
    apply (*subgoal-tac* f(b := f b + *1*) = ($\lambda a.$ f a + (*if* a=b *then 1 else 0*)))
     prefer *2*
     apply (*simp add*: *expand-fun-eq*)
    apply (*erule ssubst*)
    apply (*erule Abs-multiset-inverse* [*THEN subst*])
    apply (*erule add* [*unfolded defns*, *simplified*])
    done
qed

lemma *MCollect-preserves-multiset*:
    $M \in$ *multiset* ==> ($\lambda x.$ *if* P x *then* M x *else 0*) $\in$ *multiset*
  apply (*simp add*: *multiset-def*)
  apply (*rule finite-subset*, *auto*)
  done

lemma *count-MCollect* [*simp*]:
    count {# x:M. P x #} a = (*if* P a *then count* M a *else 0*)
  by (*simp add*: *count-def MCollect-def MCollect-preserves-multiset*)

lemma *set-of-MCollect* [*simp*]: *set-of* {# x:M. P x #} = *set-of* M $\cap$ {x. P x}
  by (*auto simp add*: *set-of-def*)

lemma *multiset-partition*: M = {# x:M. P x #} + {# x:M. $\neg$ P x #}
  by (*subst multiset-eq-conv-count-eq*, *auto*)

lemma *add-eq-conv-ex*:
  (M + {#a#} = N + {#b#}) =
    (M = N $\wedge$ a = b $\vee$ ($\exists$ K. M = K + {#b#} $\wedge$ N = K + {#a#}))
  by (*auto simp add*: *add-eq-conv-diff*)

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

1.4 Multiset 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*
  by (*simp add*: *mult1-def*)

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 (*unfold mult1-def*)
  let *?r* = λ*K a*. ∀ *b*. *b* :# *K* −−> (*b*, *a*) ∈ *r*
  let *?R* = λ*N M*. ∃ *a M0 K*. *M* = *M0* + {#*a*#} ∧ *N* = *M0* + *K* ∧ *?r K a*
  let *?case1* = *?case1* {(*N*, *M*). *?R N M*}

  assume (*N*, *M0* + {#*a*#}) ∈ {(*N*, *M*). *?R N M*}
  then have ∃ *a' M0' K*.
      M0 + {#*a*#} = *M0'* + {#*a'*#} ∧ *N* = *M0'* + *K* ∧ *?r K a'* **by** *simp*
  then show *?case1* ∨ *?case2*
  proof (*elim exE conjE*)
    fix *a' M0' K*
    assume *N*: *N* = *M0'* + *K* **and** *r*: *?r K a'*
    assume *M0* + {#*a*#} = *M0'* + {#*a'*#}
    then have *M0* = *M0'* ∧ *a* = *a'* ∨
        (∃ *K'*. *M0* = *K'* + {#*a'*#} ∧ *M0'* = *K'* + {#*a*#})
      by (*simp only*: *add-eq-conv-ex*)
    then show *?thesis*
    proof (*elim disjE conjE exE*)
      assume *M0* = *M0'* *a* = *a'*
      with *N r* **have** *?r K a* ∧ *N* = *M0* + *K* **by** *simp*
      then have *?case2* .. **then show** *?thesis* ..
    next
      fix *K'*
      assume *M0'* = *K'* + {#*a*#}
      with *N* **have** *n*: *N* = *K'* + *K* + {#*a*#} **by** (*simp add*: *union-ac*)

      assume *M0* = *K'* + {#*a'*#}
      with *r* **have** *?R* (*K'* + *K*) *M0* **by** *blast*
      with *n* **have** *?case1* **by** *simp* **then show** *?thesis* ..

```
    qed
  qed
qed

lemma all-accessible: wf r ==> ∀ M. M ∈ acc (mult1 r)
proof
  let ?R = mult1 r
  let ?W = acc ?R
  {
    fix M M0 a
    assume M0: M0 ∈ ?W
      and wf-hyp: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
      and acc-hyp: ∀ M. (M, M0) ∈ ?R --> M + {#a#} ∈ ?W
    have M0 + {#a#} ∈ ?W
    proof (rule accI [of M0 + {#a#}])
      fix N
      assume (N, M0 + {#a#}) ∈ ?R
      then have ((∃ M. (M, M0) ∈ ?R ∧ N = M + {#a#}) ∨
          (∃ K. (∀ b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K))
        by (rule less-add)
      then show N ∈ ?W
      proof (elim exE disjE conjE)
        fix M assume (M, M0) ∈ ?R and N: N = M + {#a#}
        from acc-hyp have (M, M0) ∈ ?R --> M + {#a#} ∈ ?W ..
        from this and ‹(M, M0) ∈ ?R› have M + {#a#} ∈ ?W ..
        then show N ∈ ?W by (simp only: N)
      next
        fix K
        assume N: N = M0 + K
        assume ∀ b. b :# K --> (b, a) ∈ r
        then have M0 + K ∈ ?W
        proof (induct K)
          case empty
          from M0 show M0 + {#} ∈ ?W by simp
        next
          case (add K x)
          from add.prems have (x, a) ∈ r by simp
          with wf-hyp have ∀ M ∈ ?W. M + {#x#} ∈ ?W by blast
          moreover from add have M0 + K ∈ ?W by simp
          ultimately have (M0 + K) + {#x#} ∈ ?W ..
          then show M0 + (K + {#x#}) ∈ ?W by (simp only: union-assoc)
        qed
        then show N ∈ ?W by (simp only: N)
      qed
    qed
  } note tedious-reasoning = this

  assume wf: wf r
  fix M
```

```
  show M ∈ ?W
  proof (induct M)
    show {#} ∈ ?W
    proof (rule accI)
      fix b assume (b, {#}) ∈ ?R
      with not-less-empty show b ∈ ?W by contradiction
    qed

    fix M a assume M ∈ ?W
    from wf have ∀ M ∈ ?W. M + {#a#} ∈ ?W
    proof induct
      fix a
      assume r: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
      show ∀ M ∈ ?W. M + {#a#} ∈ ?W
      proof
        fix M assume M ∈ ?W
        then show M + {#a#} ∈ ?W
          by (rule acc-induct) (rule tedious-reasoning [OF - r])
      qed
    qed
    from this and ‹M ∈ ?W› show M + {#a#} ∈ ?W ..
  qed
qed

theorem wf-mult1: wf r ==> wf (mult1 r)
  by (rule acc-wfI) (rule all-accessible)

theorem wf-mult: wf r ==> wf (mult r)
  unfolding mult-def by (rule wf-trancl) (rule wf-mult1)
```

1.4.2 Closure-free presentation

```
lemma diff-union-single-conv: a :# J ==> I + J − {#a#} = I + (J − {#a#})
  by (simp add: multiset-eq-conv-count-eq)
```

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)
  apply (unfold mult-def mult1-def set-of-def)
  apply (erule converse-trancl-induct, clarify)
   apply (rule-tac x = M0 in exI, simp, clarify)
  apply (case-tac a :# K)
   apply (rule-tac x = I in exI)
   apply (simp (no-asm))
   apply (rule-tac x = (K − {#a#}) + Ka in exI)
   apply (simp (no-asm-simp) add: union-assoc [symmetric])
   apply (drule-tac f = λM. M − {#a#} in arg-cong)
```

apply (*simp add*: *diff-union-single-conv*)
apply (*simp* (*no-asm-use*) *add*: *trans-def*)
apply *blast*
apply (*subgoal-tac a* :# *I*)
apply (*rule-tac* $x = I - \{\#a\#\}$ **in** *exI*)
apply (*rule-tac* $x = J + \{\#a\#\}$ **in** *exI*)
apply (*rule-tac* $x = K + Ka$ **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *multiset-eq-conv-count-eq split*: *nat-diff-split*)
apply (*rule conjI*)
apply (*drule-tac* $f = \lambda M.\ M - \{\#a\#\}$ **in** *arg-cong*, *simp*)
apply (*simp add*: *multiset-eq-conv-count-eq split*: *nat-diff-split*)
apply (*simp* (*no-asm-use*) *add*: *trans-def*)
apply *blast*
apply (*subgoal-tac a* :# (*M0* + {#*a*#}))
apply *simp*
apply (*simp* (*no-asm*))
done

lemma *elem-imp-eq-diff-union*: *a* :# *M* ==> $M = M - \{\#a\#\} + \{\#a\#\}$
by (*simp add*: *multiset-eq-conv-count-eq*)

lemma *size-eq-Suc-imp-eq-union*: *size M* = *Suc n* ==> $\exists\, a\ N.\ M = N + \{\#a\#\}$
apply (*erule size-eq-Suc-imp-elem* [*THEN exE*])
apply (*drule elem-imp-eq-diff-union*, *auto*)
done

lemma *one-step-implies-mult-aux*:
trans r ==>
$\forall\, I\ J\ K.$ (*size* $J = n \wedge J \neq \{\#\} \wedge (\forall\, k \in$ *set-of* $K.\ \exists\, j \in$ *set-of* $J.\ (k, j) \in r)$)
−−> $(I + K, I + J) \in$ *mult r*
apply (*induct-tac n*, *auto*)
apply (*frule size-eq-Suc-imp-eq-union*, *clarify*)
apply (*rename-tac J′*, *simp*)
apply (*erule notE*, *auto*)
apply (*case-tac* $J' = \{\#\}$)
apply (*simp add*: *mult-def*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*, *blast*)

Now we know $J' \neq \{\#\}$.

apply (*cut-tac* $M = K$ **and** $P = \lambda x.\ (x, a) \in r$ **in** *multiset-partition*)
apply (*erule-tac* $P = \forall\, k \in$ *set-of* $K.$ *?P k* **in** *rev-mp*)
apply (*erule ssubst*)
apply (*simp add*: *Ball-def*, *auto*)
apply (*subgoal-tac*
$((I + \{\#\ x : K.\ (x, a) \in r\ \#\}) + \{\#\ x : K.\ (x, a) \notin r\ \#\},$
$(I + \{\#\ x : K.\ (x, a) \in r\ \#\}) + J') \in$ *mult r*)
prefer *2*

apply *force*
apply (*simp* (*no-asm-use*) *add*: *union-assoc* [*symmetric*] *mult-def*)
apply (*erule trancl-trans*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*)
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = I + J'$ **in** *exI*)
apply (*simp add*: *union-ac*)
done

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

1.4.3 Partial-order properties

instance *multiset* :: (*type*) *ord* **..**

defs (overloaded)
less-multiset-def: $M' < M$ == $(M', M) \in$ *mult* $\{(x', x).\ x' < x\}$
le-multiset-def: $M' <= M$ == $M' = M \vee M' < (M::'a\ multiset)$

lemma *trans-base-order*: *trans* $\{(x', x).\ x' < (x::'a::order)\}$
unfolding *trans-def* **by** (*blast intro*: *order-less-trans*)

Irreflexivity.

lemma *mult-irrefl-aux*:
finite A ==> $(\forall x \in A.\ \exists y \in A.\ x < (y::'a::order)) \Longrightarrow A = \{\}$
by (*induct rule*: *finite-induct*) (*auto intro*: *order-less-trans*)

lemma *mult-less-not-refl*: $\neg\ M < (M::'a::order\ multiset)$
apply (*unfold less-multiset-def*, *auto*)
apply (*drule trans-base-order* [*THEN mult-implies-one-step*], *auto*)
apply (*drule finite-set-of* [*THEN mult-irrefl-aux* [*rule-format* (*no-asm*)]])
apply (*simp add*: *set-of-eq-empty-iff*)
done

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::order\ multiset)$ ==> R
using *insert mult-less-not-refl* **by** *fast*

Transitivity.

theorem *mult-less-trans*: $K < M$ ==> $M < N$ ==> $K < (N::'a::order\ multiset)$
unfolding *less-multiset-def mult-def* **by** (*blast intro*: *trancl-trans*)

Asymmetry.

theorem *mult-less-not-sym*: $M < N$ ==> $\neg\ N < (M::'a::order\ multiset)$
apply *auto*

apply (*rule mult-less-not-refl* [*THEN notE*])
apply (*erule mult-less-trans*, *assumption*)
done

theorem *mult-less-asym*:
$M < N ==> (\neg\ P ==> N < (M::'a::order\ multiset)) ==> P$
by (*insert mult-less-not-sym*, *blast*)

theorem *mult-le-refl* [*iff*]: $M <= (M::'a::order\ multiset)$
unfolding *le-multiset-def* **by** *auto*

Anti-symmetry.

theorem *mult-le-antisym*:
$M <= N ==> N <= M ==> M = (N::'a::order\ multiset)$
unfolding *le-multiset-def* **by** (*blast dest*: *mult-less-not-sym*)

Transitivity.

theorem *mult-le-trans*:
$K <= M ==> M <= N ==> K <= (N::'a::order\ multiset)$
unfolding *le-multiset-def* **by** (*blast intro*: *mult-less-trans*)

theorem *mult-less-le*: $(M < N) = (M <= N \wedge M \neq (N::'a::order\ multiset))$
unfolding *le-multiset-def* **by** *auto*

Partial order.

instance *multiset* :: (*order*) *order*
apply *intro-classes*
apply (*rule mult-less-le*)
apply (*rule mult-le-refl*)
apply (*erule mult-le-trans*, *assumption*)
apply (*erule mult-le-antisym*, *assumption*)
done

1.4.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in mult1\ r ==> trans\ r ==> (C + B, C + D) \in mult1\ r$
apply (*unfold mult1-def*, *auto*)
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = C + M0$ **in** *exI*)
apply (*simp add*: *union-assoc*)
done

lemma *union-less-mono2*: $B < D ==> C + B < C + (D::'a::order\ multiset)$
apply (*unfold less-multiset-def mult-def*)
apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl*)
apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl trancl-trans*)
done

lemma *union-less-mono1*: $B < D ==> B + C < D + (C::'a::order\ multiset)$
  apply (*subst union-commute* [*of B C*])
  apply (*subst union-commute* [*of D C*])
  apply (*erule union-less-mono2*)
  done

lemma *union-less-mono*:
    $A < C ==> B < D ==> A + B < C + (D::'a::order\ multiset)$
  apply (*blast intro*!: *union-less-mono1 union-less-mono2 mult-less-trans*)
  done

lemma *union-le-mono*:
    $A <= C ==> B <= D ==> A + B <= C + (D::'a::order\ multiset)$
  unfolding *le-multiset-def*
  by (*blast intro*: *union-less-mono union-less-mono1 union-less-mono2*)

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
  apply (*unfold le-multiset-def less-multiset-def*)
  apply (*case-tac* $M = \{\#\}$)
   prefer *2*
   apply (*subgoal-tac* $(\{\#\} + \{\#\}, \{\#\} + M) \in mult\ (Collect\ (split\ op\ <))$)
    prefer *2*
    apply (*rule one-step-implies-mult*)
      apply (*simp only*: *trans-def*, *auto*)
  done

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
proof −
  have $A + \{\#\} <= A + B$ **by** (*blast intro*: *union-le-mono*)
  then show *?thesis* **by** *simp*
qed

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
  by (*subst union-commute*) (*rule union-upper1*)

instance *multiset* :: (*order*) *pordered-ab-semigroup-add*
apply *intro-classes*
apply(*erule union-le-mono*[*OF mult-le-refl*])
done

1.5 Link with lists

consts
  multiset-of :: $'a\ list \Rightarrow 'a\ multiset$
primrec
  multiset-of $[] = \{\#\}$
  multiset-of $(a\ \#\ x) =$ *multiset-of* $x + \{\#\ a\ \#\}$

lemma *multiset-of-zero-iff* [*simp*]: (*multiset-of* x = {#}) = (x = [])
by (*induct* x) *auto*

lemma *multiset-of-zero-iff-right*[*simp*]: ({#} = *multiset-of* x) = (x = [])
by (*induct* x) *auto*

lemma *set-of-multiset-of* [*simp*]: *set-of* (*multiset-of* x) = *set* x
by (*induct* x) *auto*

lemma *mem-set-multiset-eq*: $x \in$ *set* *xs* = (x :# *multiset-of* *xs*)
by (*induct* *xs*) *auto*

lemma *multiset-of-append* [*simp*]:
multiset-of (*xs* @ *ys*) = *multiset-of* *xs* + *multiset-of* *ys*
by (*induct* *xs* *arbitrary*: *ys*) (*auto* *simp*: *union-ac*)

lemma *surj-multiset-of*: *surj* *multiset-of*
apply (*unfold* *surj-def*, *rule* *allI*)
apply (*rule-tac* M=y **in** *multiset-induct*, *auto*)
apply (*rule-tac* x = x # *xa* **in** *exI*, *auto*)
done

lemma *set-count-greater-0*: *set* x = {a. *count* (*multiset-of* x) a > *0*}
by (*induct* x) *auto*

lemma *distinct-count-atmost-1*:
distinct x = (! a. *count* (*multiset-of* x) a = (*if* $a \in$ *set* x *then* *1* *else* *0*))
apply (*induct* x, *simp*, *rule* *iffI*, *simp-all*)
apply (*rule* *conjI*)
apply (*simp-all* *add*: *set-of-multiset-of* [*THEN* *sym*] *del*: *set-of-multiset-of*)
apply (*erule-tac* x=a **in** *allE*, *simp*, *clarify*)
apply (*erule-tac* x=*aa* **in** *allE*, *simp*)
done

lemma *multiset-of-eq-setD*:
multiset-of *xs* = *multiset-of* *ys* $\Longrightarrow$ *set* *xs* = *set* *ys*
by (*rule*) (*auto* *simp* *add*:*multiset-eq-conv-count-eq* *set-count-greater-0*)

lemma *set-eq-iff-multiset-of-eq-distinct*:
⟦*distinct* x; *distinct* y⟧
$\Longrightarrow$ (*set* x = *set* y) = (*multiset-of* x = *multiset-of* y)
by (*auto* *simp*: *multiset-eq-conv-count-eq* *distinct-count-atmost-1*)

lemma *set-eq-iff-multiset-of-remdups-eq*:
(*set* x = *set* y) = (*multiset-of* (*remdups* x) = *multiset-of* (*remdups* y))
apply (*rule* *iffI*)
apply (*simp* *add*: *set-eq-iff-multiset-of-eq-distinct*[*THEN* *iffD1*])
apply (*drule* *distinct-remdups*[*THEN* *distinct-remdups*
[*THEN* *set-eq-iff-multiset-of-eq-distinct*[*THEN* *iffD2*]]])

apply *simp*
done

lemma *multiset-of-compl-union* [*simp*]:
$multiset\text{-}of\ [x{\leftarrow}xs.\ P\ x] + multiset\text{-}of\ [x{\leftarrow}xs.\ \neg P\ x] = multiset\text{-}of\ xs$
by (*induct xs*) (*auto simp*: *union-ac*)

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

1.6 Pointwise ordering induced by count

definition
$mset\text{-}le :: {'a}\ multiset \Rightarrow {'a}\ multiset \Rightarrow bool$ (**infix** $\leq\#$ *50*) **where**
$(A \leq\# B) = (\forall a.\ count\ A\ a \leq count\ B\ a)$
definition
$mset\text{-}less :: {'a}\ multiset \Rightarrow {'a}\ multiset \Rightarrow bool$ (**infix** $<\#$ *50*) **where**
$(A <\# B) = (A \leq\# B \wedge A \neq B)$

lemma *mset-le-refl*[*simp*]: $A \leq\# A$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-trans*: $[\![A \leq\# B;\ B \leq\# C]\!] \Longrightarrow A \leq\# C$
unfolding *mset-le-def* **by** (*fast intro*: *order-trans*)

lemma *mset-le-antisym*: $[\![A \leq\# B;\ B \leq\# A]\!] \Longrightarrow A = B$
apply (*unfold mset-le-def*)
apply (*rule multiset-eq-conv-count-eq*[*THEN iffD2*])
apply (*blast intro*: *order-antisym*)
done

lemma *mset-le-exists-conv*:
$(A \leq\# B) = (\exists C.\ B = A + C)$
apply (*unfold mset-le-def*, *rule iffI*, *rule-tac* $x = B - A$ **in** *exI*)
apply (*auto intro*: *multiset-eq-conv-count-eq* [*THEN iffD2*])
done

lemma *mset-le-mono-add-right-cancel*[*simp*]: $(A + C \leq\# B + C) = (A \leq\# B)$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-mono-add-left-cancel*[*simp*]: $(C + A \leq\# C + B) = (A \leq\# B)$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-mono-add*: $[\![A \leq\# B;\ C \leq\# D]\!] \Longrightarrow A + C \leq\# B + D$
apply (*unfold mset-le-def*)
apply *auto*
apply (*erule-tac* $x{=}a$ **in** *allE*)+
apply *auto*

done

lemma *mset-le-add-left*[*simp*]: $A \le\# A + B$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-add-right*[*simp*]: $B \le\# A + B$
unfolding *mset-le-def* **by** *auto*

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups xs*) $\le\#$ *multiset-of xs*
apply (*induct xs*)
apply *auto*
apply (*rule mset-le-trans*)
apply *auto*
done

interpretation *mset-order*:
order [*op* $\le\#$ *op* $<\#$]
by (*auto intro*: *order.intro mset-le-refl mset-le-antisym*
mset-le-trans simp: *mset-less-def*)

interpretation *mset-order-cancel-semigroup*:
pordered-cancel-ab-semigroup-add [*op* $\le\#$ *op* $<\#$ *op* $+$]
by *unfold-locales* (*erule mset-le-mono-add* [*OF mset-le-refl*])

interpretation *mset-order-semigroup-cancel*:
pordered-ab-semigroup-add-imp-le [*op* $\le\#$ *op* $<\#$ *op* $+$]
by (*unfold-locales*) *simp*

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* $R \Longrightarrow (a,b) \in R \lor a = b \Longrightarrow (ah@a\#at, bh@b\#bt) \in$ *lprod* R

lemma $(as,bs) \in$ *lprod* $R \Longrightarrow$ *length as* $=$ *length bs*
apply (*induct as bs rule*: *lprod.induct*)
apply *auto*
done

lemma $(as,\ bs) \in lprod\ R \Longrightarrow 1 \leq length\ as \wedge 1 \leq length\ bs$
  apply (*induct as bs rule*: *lprod.induct*)
  apply *auto*
  done

lemma *lprod-subset-elem*: $(as,\ bs) \in lprod\ S \Longrightarrow S \subseteq R \Longrightarrow (as,\ bs) \in lprod\ R$
  apply (*induct as bs rule*: *lprod.induct*)
  apply (*auto*)
  done

lemma *lprod-subset*: $S \subseteq R \Longrightarrow lprod\ S \subseteq lprod\ R$
  by (*auto intro*: *lprod-subset-elem*)

lemma *lprod-implies-mult*: $(as,\ bs) \in lprod\ R \Longrightarrow trans\ R \Longrightarrow$ (*multiset-of as*, *multiset-of bs*) $\in$ *mult R*
proof (*induct as bs rule*: *lprod.induct*)
  case (*lprod-single a b*)
  note *step* = *one-step-implies-mult*[
    where $r{=}R$ **and** $I{=}\{\#\}$ **and** $K{=}\{\#a\#\}$ **and** $J{=}\{\#b\#\}$, *simplified*]
  show *?case* **by** (*auto intro*: *lprod-single step*)
next
  case (*lprod-list ah at bh bt a b*)
  from *prems* **have** *transR*: *trans R* **by** *auto*
  have *as*: *multiset-of* (*ah* @ *a* # *at*) = *multiset-of* (*ah* @ *at*) + $\{\#a\#\}$ (**is** - = *?ma* + -)
    by (*simp add*: *ring-simps*)
  have *bs*: *multiset-of* (*bh* @ *b* # *bt*) = *multiset-of* (*bh* @ *bt*) + $\{\#b\#\}$ (**is** - = *?mb* + -)
    by (*simp add*: *ring-simps*)
  from *prems* **have** (*?ma*, *?mb*) $\in$ *mult R*
    by *auto*
  with *mult-implies-one-step*[*OF transR*] **have**
    $\exists I\ J\ K.$ *?mb* = $I + J \wedge$ *?ma* = $I + K \wedge J \neq \{\#\} \wedge (\forall k{\in}$*set-of* $K.\ \exists j{\in}$*set-of* $J.\ (k,\ j) \in R)$
    by *blast*
  then obtain *I J K* **where**
    decomposed: *?mb* = $I + J \wedge$ *?ma* = $I + K \wedge J \neq \{\#\} \wedge (\forall k{\in}$*set-of* $K.\ \exists j{\in}$*set-of* $J.\ (k,\ j) \in R)$
    by *blast*
  show *?case*
  proof (*cases* $a = b$)
    case *True*
    have $((I + \{\#b\#\}) + K,\ (I + \{\#b\#\}) + J) \in$ *mult R*
      apply (*rule one-step-implies-mult*[*OF transR*])
      apply (*auto simp add*: *decomposed*)
      done
    then show *?thesis*
      apply (*simp only*: *as bs*)
      apply (*simp only*: *decomposed True*)

    apply (*simp add*: *ring-simps*)
    done
  next
   case *False*
   from *False lprod-list* **have** *False*: $(a, b) \in R$ **by** *blast*
   have $(I + (K + \{\#a\#\}), I + (J + \{\#b\#\})) \in$ *mult R*
    apply (*rule one-step-implies-mult*[*OF transR*])
    apply (*auto simp add*: *False decomposed*)
    done
   then show *?thesis*
    apply (*simp only*: *as bs*)
    apply (*simp only*: *decomposed*)
    apply (*simp add*: *ring-simps*)
    done
  qed
qed

lemma *wf-lprod*[*recdef-wf*,*simp*,*intro*]:
  assumes *wf-R*: *wf R*
  shows *wf* (*lprod R*)
proof −
  have *subset*: *lprod* (*R*ˆ+) ⊆ *inv-image* (*mult* (*R*ˆ+)) *multiset-of*
   by (*auto simp add*: *lprod-implies-mult trans-trancl*)
  note *lprodtrancl* = *wf-subset*[*OF wf-inv-image*[**where** *r*=*mult* (*R*ˆ+) **and** *f*=*multiset-of*,

   OF wf-mult[*OF wf-trancl*[*OF wf-R*]]], *OF subset*]
  note *lprod* = *wf-subset*[*OF lprodtrancl*, **where** *p*=*lprod R*, *OF lprod-subset*, *simplified*]
  show *?thesis* **by** (*auto intro*: *lprod*)
qed

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

lemma *lprod-2-3*: $(a, b) \in R \Longrightarrow ([a, c], [b, c]) \in$ *lprod R*
  by (*auto intro*: *lprod-list*[**where** *a*=*c* **and** *b*=*c* **and**
   ah = [*a*] **and** *at* = [] **and** *bh*=[*b*] **and** *bt*=[], *simplified*])

lemma *lprod-2-4*: $(a, b) \in R \Longrightarrow ([c, a], [c, b]) \in$ *lprod R*
  by (*auto intro*: *lprod-list*[**where** *a*=*c* **and** *b*=*c* **and**
   ah = [] **and** *at* = [*a*] **and** *bh*=[] **and** *bt*=[*b*], *simplified*])

lemma *lprod-2-1*: $(a, b) \in R \Longrightarrow ([c, a], [b, c]) \in$ *lprod R*
  by (*auto intro*: *lprod-list*[**where** *a*=*c* **and** *b*=*c* **and**
   ah = [] **and** *at* = [*a*] **and** *bh*=[*b*] **and** *bt*=[], *simplified*])

lemma *lprod-2-2*: $(a,\ b) \in R \Longrightarrow ([a,\ c],\ [c,\ b]) \in$ *lprod R*
  by (*auto intro*: *lprod-list*[**where** $a=c$ **and** $b=c$ **and**
    $ah = [a]$ **and** $at = []$ **and** $bh=[]$ **and** $bt=[b]$, *simplified*])

lemma [*recdef-wf*, *simp*, *intro*]:
  assumes *wfR*: *wf R* **shows** *wf* (*gprod-2-1 R*)
proof −
  have *gprod-2-1 R* $\subseteq$ *inv-image* (*lprod R*) (λ (a,b). $[a,b]$)
    by (*auto simp add*: *gprod-2-1-def lprod-2-1 lprod-2-2*)
  with *wfR* **show** *?thesis*
    by (*rule-tac wf-subset*, *auto*)
qed

lemma [*recdef-wf*, *simp*, *intro*]:
  assumes *wfR*: *wf R* **shows** *wf* (*gprod-2-2 R*)
proof −
  have *gprod-2-2 R* $\subseteq$ *inv-image* (*lprod R*) (λ (a,b). $[a,b]$)
    by (*auto simp add*: *gprod-2-2-def lprod-2-3 lprod-2-4*)
  with *wfR* **show** *?thesis*
    by (*rule-tac wf-subset*, *auto*)
qed

lemma *lprod-3-1*: **assumes** $(x',\ x) \in R$ **shows** $([y,\ z,\ x'],\ [x,\ y,\ z]) \in$ *lprod R*
  apply (*rule lprod-list*[**where** $a=y$ **and** $b=y$ **and** $ah=[]$ **and** $at=[z,x']$ **and** $bh=[x]$
and $bt=[z]$, *simplified*])
  apply (*auto simp add*: *lprod-2-1 prems*)
  done

lemma *lprod-3-2*: **assumes** $(z',z) \in R$ **shows** $([z',\ x,\ y],\ [x,y,z]) \in$ *lprod R*
  apply (*rule lprod-list*[**where** $a=y$ **and** $b=y$ **and** $ah=[z',x]$ **and** $at=[]$ **and** $bh=[x]$
and $bt=[z]$, *simplified*])
  apply (*auto simp add*: *lprod-2-2 prems*)
  done

lemma *lprod-3-3*: **assumes** *xr*: $(xr,\ x) \in R$ **shows** $([xr,\ y,\ z],\ [x,\ y,\ z]) \in$ *lprod R*
  apply (*rule lprod-list*[**where** $a=y$ **and** $b=y$ **and** $ah=[xr]$ **and** $at=[z]$ **and** $bh=[x]$
and $bt=[z]$, *simplified*])
  apply (*simp add*: *xr lprod-2-3*)
  done

lemma *lprod-3-4*: **assumes** *yr*: $(yr,\ y) \in R$ **shows** $([x,\ yr,\ z],\ [x,\ y,\ z]) \in$ *lprod R*
  apply (*rule lprod-list*[**where** $a=x$ **and** $b=x$ **and** $ah=[]$ **and** $at=[yr,z]$ **and** $bh=[]$
and $bt=[y,z]$, *simplified*])
  apply (*simp add*: *yr lprod-2-3*)
  done

lemma *lprod-3-5*: **assumes** *zr*: $(zr,\ z) \in R$ **shows** $([x,\ y,\ zr],\ [x,\ y,\ z]) \in$ *lprod R*
  apply (*rule lprod-list*[**where** $a=x$ **and** $b=x$ **and** $ah=[]$ **and** $at=[y,zr]$ **and** $bh=[]$
and $bt=[y,z]$, *simplified*])

apply (*simp add*: *zr lprod-2-4*)
done

lemma *lprod-3-6*: **assumes** y': $(y', y) \in R$ **shows** $([x, z, y'], [x, y, z]) \in$ *lprod R*
apply (*rule lprod-list*[**where** a=z **and** b=z **and** ah=$[x]$ **and** at=$[y']$ **and** bh=$[x,y]$ **and** bt=[], *simplified*])
apply (*simp add*: y' *lprod-2-4*)
done

lemma *lprod-3-7*: **assumes** z': $(z',z) \in R$ **shows** $([x, z', y], [x, y, z]) \in$ *lprod R*
apply (*rule lprod-list*[**where** a=y **and** b=y **and** ah=$[x, z']$ **and** at=[] **and** bh=$[x]$ **and** bt=$[z]$, *simplified*])
apply (*simp add*: z' *lprod-2-4*)
done

constdefs
perm :: $('a \Rightarrow 'a) \Rightarrow 'a$ *set* $\Rightarrow$ *bool*
perm f A $\equiv$ *inj-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*))
oops

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*
oops

end

theory *MainZF*
imports *Zet LProd*
begin
end

theory *Games*
imports *MainZF*
begin

constdefs
fixgames :: *ZF set* $\Rightarrow$ *ZF set*
fixgames A $\equiv$ { *Opair l r* | *l r*. *explode l* $\subseteq$ *A* & *explode r* $\subseteq$ *A*}
games-lfp :: *ZF set*

games-lfp ≡ *lfp fixgames*
games-gfp :: *ZF set*
games-gfp ≡ *gfp fixgames*

lemma *mono-fixgames*: *mono* (*fixgames*)
apply (*auto simp add*: *mono-def fixgames-def*)
apply (*rule-tac x=l* **in** *exI*)
apply (*rule-tac x=r* **in** *exI*)
apply *auto*
done

lemma *games-lfp-unfold*: *games-lfp* = *fixgames games-lfp*
by (*auto simp add*: *def-lfp-unfold games-lfp-def mono-fixgames*)

lemma *games-gfp-unfold*: *games-gfp* = *fixgames games-gfp*
by (*auto simp add*: *def-gfp-unfold games-gfp-def mono-fixgames*)

lemma *games-lfp-nonempty*: *Opair Empty Empty* ∈ *games-lfp*
proof −
have *fixgames* {} ⊆ *games-lfp*
apply (*subst games-lfp-unfold*)
apply (*simp add*: *mono-fixgames*[*simplified mono-def*, *rule-format*])
done
moreover have *fixgames* {} = {*Opair Empty Empty*}
by (*simp add*: *fixgames-def explode-Empty*)
finally show *?thesis*
by *auto*
qed

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 −
have *games-lfp* ⊆ *fixgames games-lfp*
by (*simp add*: *games-lfp-unfold*[*symmetric*])
then show *?thesis*
by (*simp add*: *games-gfp-def gfp-upperbound*)
qed

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 −
from *g fixgames* **have** *g* ∈ *fixgames games* **by** *auto*
then have ∃ *l r*. *g* = *Opair l r* ∧ *explode l* ⊆ *games* ∧ *explode r* ⊆ *games*
by (*simp add*: *fixgames-def*)
then obtain *l* **where** ∃ *r*. *g* = *Opair l r* ∧ *explode l* ⊆ *games* ∧ *explode r* ⊆ *games* **..**
then obtain *r* **where** *lr*: *g* = *Opair l r* ∧ *explode l* ⊆ *games* ∧ *explode r* ⊆ *games* **..**
with *opt* **show** *?thesis*
by (*auto intro*: *Elem-explode-in simp add*: *left-option-def right-option-def Fst Snd*)
qed

lemma *option2elem*: (*opt*,*g*) ∈ *is-option-of* ⟹ ∃ *u v*. *Elem opt u* ∧ *Elem u v* ∧ *Elem v g*
apply (*simp add*: *is-option-of-def*)
apply (*subgoal-tac* (*g* ∈ *games-gfp*) = (*g* ∈ (*fixgames games-gfp*)))
prefer *2*
apply (*simp add*: *games-gfp-unfold*[*symmetric*])
apply (*auto simp add*: *fixgames-def left-option-def right-option-def Fst Snd*)
apply (*rule-tac x*=*l* **in** *exI*, *insert Elem-Opair-exists*, *blast*)
apply (*rule-tac x*=*r* **in** *exI*, *insert Elem-Opair-exists*, *blast*)
done

lemma *is-option-of-subset-is-Elem-of*: *is-option-of* ⊆ (*is-Elem-of*ˆ+)
proof −
{
fix *opt*
fix *g*
assume (*opt*, *g*) ∈ *is-option-of*
then have ∃ *u v*. (*opt*, *u*) ∈ (*is-Elem-of*ˆ+) ∧ (*u*,*v*) ∈ (*is-Elem-of*ˆ+) ∧ (*v*,*g*) ∈ (*is-Elem-of*ˆ+)
apply −
apply (*drule option2elem*)
apply (*auto simp add*: *r-into-trancl′ is-Elem-of-def*)
done
then have (*opt*, *g*) ∈ (*is-Elem-of*ˆ+)
by (*blast intro*: *trancl-into-rtrancl trancl-rtrancl-trancl*)
}
then show *?thesis* **by** *auto*
qed

lemma *wfzf-is-option-of*: *wfzf is-option-of*
proof −
have *wfzf* (*is-Elem-of*ˆ+) **by** (*simp add*: *wfzf-trancl wfzf-is-Elem-of*)
then show *?thesis*
apply (*rule wfzf-subset*)

```
    apply (rule is-option-of-subset-is-Elem-of)
    done
  qed

lemma games-gfp-imp-lfp: g ∈ games-gfp ⟶ g ∈ games-lfp
proof -
  have unfold-gfp: ⋀ x. x ∈ games-gfp ⟹ x ∈ (fixgames games-gfp)
    by (simp add: games-gfp-unfold[symmetric])
  have unfold-lfp: ⋀ x. (x ∈ games-lfp) = (x ∈ (fixgames games-lfp))
    by (simp add: games-lfp-unfold[symmetric])
  show ?thesis
    apply (rule wf-induct[OF wfzf-implies-wf[OF wfzf-is-option-of]])
    apply (auto simp add: is-option-of-def)
    apply (drule-tac unfold-gfp)
    apply (simp add: fixgames-def)
    apply (auto simp add: left-option-def Fst right-option-def Snd)
    apply (subgoal-tac explode l ⊆ games-lfp)
    apply (subgoal-tac explode r ⊆ games-lfp)
    apply (subst unfold-lfp)
    apply (auto simp add: fixgames-def)
    apply (simp-all add: explode-Elem Elem-explode-in)
    done
qed

theorem games-lfp-eq-gfp: games-lfp = games-gfp
  apply (auto simp add: games-gfp-imp-lfp)
  apply (insert games-lfp-subset-gfp)
  apply auto
  done

theorem unique-games: (g = fixgames g) = (g = games-lfp)
proof -
  {
    fix g
    assume g: g = fixgames g
    from g have fixgames g ⊆ g by auto
    then have l:games-lfp ⊆ g
      by (simp add: games-lfp-def lfp-lowerbound)
    from g have g ⊆ fixgames g by auto
    then have u:g ⊆ games-gfp
      by (simp add: games-gfp-def gfp-upperbound)
    from l u games-lfp-eq-gfp[symmetric] have g = games-lfp
      by auto
  }
  note games = this
  show ?thesis
    apply (rule iff[rule-format])
    apply (erule games)
    apply (simp add: games-lfp-unfold[symmetric])
```

done
qed

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*
apply (*rule games-option-stable*[**where** *g*=*g*])
apply (*simp add*: *games-lfp-unfold*[*symmetric*])
apply (*simp-all add*: *prems*)
done

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 −
from *hyp* **have** *g-game*: *g* $\in$ *games-lfp*
by (*simp add*: *is-option-of-def games-lfp-eq-gfp*)
from *hyp* **have** *left-option g opt* $\vee$ *right-option g opt*
by (*auto simp add*: *is-option-of-def*)
with *g-game games-lfp-option-stable*[*OF g-game*, *OF this*] **show** *?thesis*
by *auto*
qed

lemma *games-lfp-represent*: *x* $\in$ *games-lfp* $\Longrightarrow \exists$ *l r*. *x* = *Opair l r*
apply (*rule exI*[**where** *x*=*Fst x*])
apply (*rule exI*[**where** *x*=*Snd x*])
apply (*subgoal-tac x* $\in$ (*fixgames games-lfp*))
apply (*simp add*: *fixgames-def*)
apply (*auto simp add*: *Fst Snd*)
apply (*simp add*: *games-lfp-unfold*[*symmetric*])
done

typedef *game* = *games-lfp*
by (*blast intro*: *games-lfp-nonempty*)

constdefs
left-options :: *game* $\Rightarrow$ *game zet*
left-options g $\equiv$ *zimage Abs-game* (*zexplode* (*Fst* (*Rep-game g*)))
right-options :: *game* $\Rightarrow$ *game zet*
right-options g $\equiv$ *zimage Abs-game* (*zexplode* (*Snd* (*Rep-game g*)))
options :: *game* $\Rightarrow$ *game zet*
options g $\equiv$ *zunion* (*left-options g*) (*right-options g*)
Game :: *game zet* $\Rightarrow$ *game zet* $\Rightarrow$ *game*
Game L R $\equiv$ *Abs-game* (*Opair* (*zimplode* (*zimage Rep-game L*)) (*zimplode* (*zimage Rep-game R*)))

lemma *Repl-Rep-game-Abs-game*: $\forall$ *e*. *Elem e z* $\longrightarrow$ *e* $\in$ *games-lfp* $\Longrightarrow$ *Repl z* (*Rep-game o Abs-game*) = *z*

apply (*subst Ext*)
apply (*simp add*: *Repl*)
apply *auto*
apply (*subst Abs-game-inverse*, *simp-all add*: *game-def*)
apply (*rule-tac x=za* **in** *exI*)
apply (*subst Abs-game-inverse*, *simp-all add*: *game-def*)
done

lemma *game-split*: *g* = *Game* (*left-options g*) (*right-options g*)
proof −
have ∃ *l r*. *Rep-game g* = *Opair l r*
apply (*insert Rep-game*[*of g*])
apply (*simp add*: *game-def games-lfp-represent*)
done
then obtain *l r* **where** *lr*: *Rep-game g* = *Opair l r* **by** *auto*
have *partizan-g*: *Rep-game g* ∈ *games-lfp*
apply (*insert Rep-game*[*of g*])
apply (*simp add*: *game-def*)
done
have ∀ *e*. *Elem e l* ⟶ *left-option* (*Rep-game g*) *e*
by (*simp add*: *lr left-option-def Fst*)
then have *partizan-l*: ∀ *e*. *Elem e l* ⟶ *e* ∈ *games-lfp*
apply *auto*
apply (*rule games-lfp-option-stable*[**where** *g*=*Rep-game g*, *OF partizan-g*])
apply *auto*
done
have ∀ *e*. *Elem e r* ⟶ *right-option* (*Rep-game g*) *e*
by (*simp add*: *lr right-option-def Snd*)
then have *partizan-r*: ∀ *e*. *Elem e r* ⟶ *e* ∈ *games-lfp*
apply *auto*
apply (*rule games-lfp-option-stable*[**where** *g*=*Rep-game g*, *OF partizan-g*])
apply *auto*
done
let *?L* = *zimage* (*Abs-game*) (*zexplode l*)
let *?R* = *zimage* (*Abs-game*) (*zexplode r*)
have *L*:*?L* = *left-options g*
by (*simp add*: *left-options-def lr Fst*)
have *R*:*?R* = *right-options g*
by (*simp add*: *right-options-def lr Snd*)
have *g* = *Game ?L ?R*
apply (*simp add*: *Game-def Rep-game-inject*[*symmetric*] *comp-zimage-eq zimage-zexplode-eq zimplode-zexplode*)
apply (*simp add*: *Repl-Rep-game-Abs-game partizan-l partizan-r*)
apply (*subst Abs-game-inverse*)
apply (*simp-all add*: *lr*[*symmetric*] *Rep-game*)
done
then show *?thesis*
by (*simp add*: *L R*)
qed

```
lemma Opair-in-games-lfp:
  assumes l: explode l ⊆ games-lfp
  and r: explode r ⊆ games-lfp
  shows Opair l r ∈ games-lfp
proof −
  note f = unique-games[of games-lfp, simplified]
  show ?thesis
    apply (subst f)
    apply (simp add: fixgames-def)
    apply (rule exI[where x=l])
    apply (rule exI[where x=r])
    apply (auto simp add: l r)
    done
qed

lemma left-options[simp]: left-options (Game l r) = l
  apply (simp add: left-options-def Game-def)
  apply (subst Abs-game-inverse)
  apply (simp add: game-def)
  apply (rule Opair-in-games-lfp)
 apply (auto simp add: explode-Elem Elem-zimplode zimage-iff Rep-game[simplified
game-def])
  apply (simp add: Fst zexplode-zimplode comp-zimage-eq)
  apply (simp add: zet-ext-eq zimage-iff Rep-game-inverse)
  done

lemma right-options[simp]: right-options (Game l r) = r
  apply (simp add: right-options-def Game-def)
  apply (subst Abs-game-inverse)
  apply (simp add: game-def)
  apply (rule Opair-in-games-lfp)
 apply (auto simp add: explode-Elem Elem-zimplode zimage-iff Rep-game[simplified
game-def])
  apply (simp add: Snd zexplode-zimplode comp-zimage-eq)
  apply (simp add: zet-ext-eq zimage-iff Rep-game-inverse)
  done

lemma Game-ext: (Game l1 r1 = Game l2 r2) = ((l1 = l2) ∧ (r1 = r2))
  apply auto
  apply (subst left-options[where l=l1 and r=r1,symmetric])
  apply (subst left-options[where l=l2 and r=r2,symmetric])
  apply simp
  apply (subst right-options[where l=l1 and r=r1,symmetric])
  apply (subst right-options[where l=l2 and r=r2,symmetric])
  apply simp
  done

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*)
apply (*auto simp add*: *option-of-def*)
apply (*subst Abs-game-inverse*)
apply (*simp add*: *is-option-of-imp-games game-def*)
apply (*subst Abs-game-inverse*)
apply (*simp add*: *is-option-of-imp-games game-def*)
apply *simp*
apply (*auto simp add*: *Bex-def image-def*)
apply (*rule exI*[**where** *x*=*Rep-game option*])
apply (*rule exI*[**where** *x*=*Rep-game g*])
apply (*simp add*: *Rep-game-inverse*)
done

lemma *wf-is-option-of*: *wf is-option-of*
apply (*rule wfzf-implies-wf*)
apply (*simp add*: *wfzf-is-option-of*)
done

lemma *wf-option-of*[*recdef-wf*, *simp*, *intro*]: *wf option-of*
proof −
have *option-of*: *option-of* = *inv-image is-option-of Rep-game*
apply (*rule set-ext*)
apply (*case-tac x*)
by (*simp add*: *option-to-is-option-of*)
show *?thesis*
apply (*simp add*: *option-of*)
apply (*auto intro*: *wf-inv-image wf-is-option-of*)
done
qed

lemma *right-option-is-option*[*simp*, *intro*]: *zin x* (*right-options g*) ⟹ *zin x* (*options g*)
by (*simp add*: *options-def zunion*)

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

lemma *zin-options*[*simp*, *intro*]: *zin x* (*options g*) ⟹ (*x*, *g*) ∈ *option-of*
apply (*simp add*: *options-def zunion left-options-def right-options-def option-of-def*
image-def is-option-of-def zimage-iff zin-zexplode-eq)
apply (*cases g*)
apply (*cases x*)
apply (*auto simp add*: *Abs-game-inverse games-lfp-eq-gfp*[*symmetric*] *game-def*

right-option-def[*symmetric*] *left-option-def*[*symmetric*])
done

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*
apply (*induct g rule*: *neg-game.induct*)
apply (*subst neg-game.simps*)+
apply (*simp add*: *right-options left-options comp-zimage-eq*)
apply (*subgoal-tac zimage* (*neg-game o neg-game*) (*left-options g*) = *left-options g*)
apply (*subgoal-tac zimage* (*neg-game o neg-game*) (*right-options g*) = *right-options g*)
apply (*auto simp add*: *game-split*[*symmetric*])
apply (*auto simp add*: *zet-ext-eq zimage-iff*)
done

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-def*: *ge-game* (*G*, *H*) = (∀ *x*. (*zin x* (*right-options G*) ⟶ ¬ *ge-game* (*H*, *x*)) ∧ (*zin x* (*left-options H*) ⟶ ¬ *ge-game* (*x*, *G*)))
apply (*subst ge-game.simps*[**where** *G*=*G* **and** *H*=*H*])
apply (*auto*)
done

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 (*induct x rule*: *wf-induct*[*OF wf-option-of*])
case (*1 g*)

```
  {
    fix y
    assume y: zin y (right-options g)
    have ¬ ge-game (g, y)
    proof −
      have (y, g) ∈ option-of by (auto intro: y)
      with 1 have ge-game (y, y) by auto
      with y show ?thesis by (subst ge-game-def, auto)
    qed
  }
  note right = this
  {
    fix y
    assume y: zin y (left-options g)
    have ¬ ge-game (y, g)
    proof −
      have (y, g) ∈ option-of by (auto intro: y)
      with 1 have ge-game (y, y) by auto
      with y show ?thesis by (subst ge-game-def, auto)
    qed
  }
  note left = this
  from left right show ?case
    by (auto, subst ge-game-def, auto)
qed

lemma ge-game-refl: ge-game (x,x) by (simp add: ge-game-leftright-refl)

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 (induct x rule: wf-induct[OF wf-option-of])
  case (1 g)
  show ?case
  proof (auto)
    {case (goal1 y)
      from goal1 have (y, g) ∈ option-of by (auto)
      with 1 have ge-game (y, y) by auto
      with goal1 have ¬ ge-game (g, y)
        by (subst ge-game-def, auto)
      with goal1 show ?case by auto}
    note right = this
    {case (goal2 y)
      from goal2 have (y, g) ∈ option-of by (auto)
      with 1 have ge-game (y, y) by auto
      with goal2 have ¬ ge-game (y, g)
        by (subst ge-game-def, auto)
      with goal2 show ?case by auto}
    note left = this
    {case goal3
```

from *left right* **show** *?case*
by (*subst ge-game-def*, *auto*)
}
qed
qed

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*)
by (*auto simp add*: *eq-game-def*)

lemma *eq-game-refl*: *eq-game G G*
by (*simp add*: *ge-game-refl eq-game-def*)

lemma *induct-game*: ($\bigwedge x$. ∀ *y*. (*y*, *x*) ∈ *lprod option-of* ⟶ *P y* ⟹ *P x*) ⟹ *P a*
by (*erule wf-induct*[*OF wf-lprod*[*OF wf-option-of*]])

lemma *ge-game-trans*:
assumes *ge-game* (*x*, *y*) *ge-game* (*y*, *z*)
shows *ge-game* (*x*, *z*)
proof −
{
fix *a*
have ∀ *x y z*. *a* = [*x*,*y*,*z*] ⟶ *ge-game* (*x*,*y*) ⟶ *ge-game* (*y*,*z*) ⟶ *ge-game* (*x*, *z*)
proof (*induct a rule*: *induct-game*)
case (*1 a*)
show *?case*
proof (*rule allI* | *rule impI*)+
case (*goal1 x y z*)
show *?case*
proof −
{ fix *xr*
assume *xr*:*zin xr* (*right-options x*)
assume *ge-game* (*z*, *xr*)
have *ge-game* (*y*, *xr*)
apply (*rule 1*[*rule-format*, **where** *y*=[*y*,*z*,*xr*]])
apply (*auto intro*: *xr lprod-3-1 simp add*: *prems*)
done
moreover from *xr* **have** ¬ *ge-game* (*y*, *xr*)
by (*simp add*: *goal1*(*2*)[*simplified ge-game-def*[*of x y*], *rule-format*, *of xr*, *simplified xr*])
ultimately have *False* **by** *auto*
}
note *xr* = *this*
{ fix *zl*

assume *zl*:*zin zl* (*left-options z*)
assume *ge-game* (*zl*, *x*)
have *ge-game* (*zl*, *y*)
apply (*rule 1*[*rule-format*, **where** *y*=[*zl*,*x*,*y*]])
apply (*auto intro*: *zl lprod-3-2 simp add*: *prems*)
done
moreover from *zl* **have** ¬ *ge-game* (*zl*, *y*)
by (*simp add*: *goal1*(*3*)[*simplified ge-game-def*[*of y z*], *rule-format*, *of zl*, *simplified zl*])
ultimately have *False* **by** *auto*
}
note *zl* = *this*
show *?thesis*
by (*auto simp add*: *ge-game-def*[*of x z*] *intro*: *xr zl*)
qed
qed
qed
}
note *trans* = *this*[*of* [*x*, *y*, *z*], *simplified*, *rule-format*]
with *prems* **show** *?thesis* **by** *blast*
qed

lemma *eq-game-trans*: *eq-game a b* ⟹ *eq-game b c* ⟹ *eq-game a c*
by (*auto simp add*: *eq-game-def intro*: *ge-game-trans*)

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 (*induct G H rule*: *plus-game.induct*)
case (*1 G H*)
show *?case*
by (*auto simp add*:
plus-game.simps[**where** *G*=*G* **and** *H*=*H*]
plus-game.simps[**where** *G*=*H* **and** *H*=*G*]

Game-ext zet-ext-eq zunion zimage-iff prems)
qed

lemma *game-ext-eq*: (G = H) = (*left-options* G = *left-options* H ∧ *right-options* G = *right-options* H)
proof −
have (G = H) = (*Game* (*left-options* G) (*right-options* G) = *Game* (*left-options* H) (*right-options* H))
by (*simp add*: *game-split*[*symmetric*])
then show *?thesis* **by** *auto*
qed

lemma *left-zero-game*[*simp*]: *left-options* (*zero-game*) = *zempty*
by (*simp add*: *zero-game-def*)

lemma *right-zero-game*[*simp*]: *right-options* (*zero-game*) = *zempty*
by (*simp add*: *zero-game-def*)

lemma *plus-game-zero-right*[*simp*]: *plus-game* (G, *zero-game*) = G
proof −
{
fix G H
have H = *zero-game* ⟶ *plus-game* (G, H) = G
proof (*induct G H rule*: *plus-game.induct*, *rule impI*)
case (*goal1 G H*)
note *induct-hyp* = *prems*[*simplified goal1*, *simplified*] **and** *prems*
show *?case*
apply (*simp only*: *plus-game.simps*[**where** G=G **and** H=H])
apply (*simp add*: *game-ext-eq prems*)
apply (*auto simp add*:
zimage-cong[**where** f = λ g. *plus-game* (g, *zero-game*) **and** g = *id*]
induct-hyp)
done
qed
}
then show *?thesis* **by** *auto*
qed

lemma *plus-game-zero-left*: *plus-game* (*zero-game*, G) = G
by (*simp add*: *plus-game-comm*)

lemma *left-imp-options*[*simp*]: *zin opt* (*left-options* g) ⟹ *zin opt* (*options* g)
by (*simp add*: *options-def zunion*)

lemma *right-imp-options*[*simp*]: *zin opt* (*right-options* g) ⟹ *zin opt* (*options* g)
by (*simp add*: *options-def zunion*)

lemma *left-options-plus*:
left-options (*plus-game* (u, v)) = *zunion* (*zimage* (λg. *plus-game* (g, v)) (*left-options*

u)) (*zimage* (*λh. plus-game* (*u*, *h*)) (*left-options v*))
  by (*subst plus-game.simps*, *simp*)

lemma *right-options-plus*:
  right-options (*plus-game* (*u*, *v*)) = *zunion* (*zimage* (*λg. plus-game* (*g*, *v*)) (*right-options u*)) (*zimage* (*λh. plus-game* (*u*, *h*)) (*right-options v*))
  by (*subst plus-game.simps*, *simp*)

lemma *left-options-neg*: *left-options* (*neg-game u*) = *zimage neg-game* (*right-options u*)
  by (*subst neg-game.simps*, *simp*)

lemma *right-options-neg*: *right-options* (*neg-game u*) = *zimage neg-game* (*left-options u*)
  by (*subst neg-game.simps*, *simp*)

lemma *plus-game-assoc*: *plus-game* (*plus-game* (*F*, *G*), *H*) = *plus-game* (*F*, *plus-game* (*G*, *H*))
proof −
  {
    fix *a*
    have ∀ *F G H*. *a* = [*F*, *G*, *H*] ⟶ *plus-game* (*plus-game* (*F*, *G*), *H*) = *plus-game* (*F*, *plus-game* (*G*, *H*))
    proof (*induct a rule*: *induct-game*, (*rule impI* | *rule allI*)+)
      case (*goal1 x F G H*)
      let *?L* = *plus-game* (*plus-game* (*F*, *G*), *H*)
      let *?R* = *plus-game* (*F*, *plus-game* (*G*, *H*))
      note *options-plus* = *left-options-plus right-options-plus*
      {
        fix *opt*
        note *hyp* = *goal1*(*1*)[*simplified goal1*(*2*), *rule-format*]
        have *F*: *zin opt* (*options F*) ⟹ *plus-game* (*plus-game* (*opt*, *G*), *H*) = *plus-game* (*opt*, *plus-game* (*G*, *H*))
          by (*blast intro*: *hyp lprod-3-3*)
        have *G*: *zin opt* (*options G*) ⟹ *plus-game* (*plus-game* (*F*, *opt*), *H*) = *plus-game* (*F*, *plus-game* (*opt*, *H*))
          by (*blast intro*: *hyp lprod-3-4*)
        have *H*: *zin opt* (*options H*) ⟹ *plus-game* (*plus-game* (*F*, *G*), *opt*) = *plus-game* (*F*, *plus-game* (*G*, *opt*))
          by (*blast intro*: *hyp lprod-3-5*)
        note *F* **and** *G* **and** *H*
      }
      note *induct-hyp* = *this*
      have *left-options ?L* = *left-options ?R* ∧ *right-options ?L* = *right-options ?R*
        by (*auto simp add*:
          plus-game.simps[**where** *G*=*plus-game* (*F*,*G*) **and** *H*=*H*]
          plus-game.simps[**where** *G*=*F* **and** *H*=*plus-game* (*G*,*H*)]
          zet-ext-eq zunion zimage-iff options-plus
          induct-hyp left-imp-options right-imp-options)

```
      then show ?case
        by (simp add: game-ext-eq)
    qed
  }
  then show ?thesis by auto
qed

lemma neg-plus-game: neg-game (plus-game (G, H)) = plus-game(neg-game G,
neg-game H)
proof (induct G H rule: plus-game.induct)
  case (1 G H)
  note opt-ops =
    left-options-plus right-options-plus
    left-options-neg right-options-neg
  show ?case
    by (auto simp add: opt-ops
      neg-game.simps[of plus-game (G,H)]
      plus-game.simps[of neg-game G neg-game H]
      Game-ext zet-ext-eq zunion zimage-iff prems)
qed

lemma eq-game-plus-inverse: eq-game (plus-game (x, neg-game x)) zero-game
proof (induct x rule: wf-induct[OF wf-option-of])
  case (goal1 x)
  { fix y
    assume zin y (options x)
    then have eq-game (plus-game (y, neg-game y)) zero-game
      by (auto simp add: prems)
  }
  note ihyp = this
  {
    fix y
    assume y: zin y (right-options x)
    have ¬ (ge-game (zero-game, plus-game (y, neg-game x)))
      apply (subst ge-game.simps, simp)
      apply (rule exI[where x=plus-game (y, neg-game y)])
      apply (auto simp add: ihyp[of y, simplified y right-imp-options eq-game-def])
    apply (auto simp add: left-options-plus left-options-neg zunion zimage-iff intro:
prems)
      done
  }
  note case1 = this
  {
    fix y
    assume y: zin y (left-options x)
    have ¬ (ge-game (zero-game, plus-game (x, neg-game y)))
      apply (subst ge-game.simps, simp)
      apply (rule exI[where x=plus-game (y, neg-game y)])
      apply (auto simp add: ihyp[of y, simplified y left-imp-options eq-game-def])
```

apply (*auto simp add*: *left-options-plus zunion zimage-iff intro*: *prems*)
done
}
note *case2* = *this*
{
fix *y*
assume *y*: *zin y* (*left-options x*)
have ¬ (*ge-game* (*plus-game* (*y*, *neg-game x*), *zero-game*))
apply (*subst ge-game.simps*, *simp*)
apply (*rule exI*[**where** *x*=*plus-game* (*y*, *neg-game y*)])
apply (*auto simp add*: *ihyp*[*of y*, *simplified y left-imp-options eq-game-def*])
apply (*auto simp add*: *right-options-plus right-options-neg zunion zimage-iff intro*: *prems*)
done
}
note *case3* = *this*
{
fix *y*
assume *y*: *zin y* (*right-options x*)
have ¬ (*ge-game* (*plus-game* (*x*, *neg-game y*), *zero-game*))
apply (*subst ge-game.simps*, *simp*)
apply (*rule exI*[**where** *x*=*plus-game* (*y*, *neg-game y*)])
apply (*auto simp add*: *ihyp*[*of y*, *simplified y right-imp-options eq-game-def*])
apply (*auto simp add*: *right-options-plus zunion zimage-iff intro*: *prems*)
done
}
note *case4* = *this*
show *?case*
apply (*simp add*: *eq-game-def*)
apply (*simp add*: *ge-game.simps*[*of plus-game* (*x*, *neg-game x*) *zero-game*])
apply (*simp add*: *ge-game.simps*[*of zero-game plus-game* (*x*, *neg-game x*)])
apply (*simp add*: *right-options-plus left-options-plus right-options-neg left-options-neg zunion zimage-iff*)
apply (*auto simp add*: *case1 case2 case3 case4*)
done
qed

lemma *ge-plus-game-left*: *ge-game* (*y*,*z*) = *ge-game*(*plus-game* (*x*, *y*), *plus-game* (*x*, *z*))
proof −
{ **fix** *a*
have ∀ *x y z*. *a* = [*x*,*y*,*z*] ⟶ *ge-game* (*y*,*z*) = *ge-game*(*plus-game* (*x*, *y*), *plus-game* (*x*, *z*))
proof (*induct a rule*: *induct-game*, (*rule impI* | *rule allI*)+)
case (*goal1 a x y z*)
note *induct-hyp* = *goal1*(*1*)[*rule-format*, *simplified goal1*(*2*)]
{
assume *hyp*: *ge-game*(*plus-game* (*x*, *y*), *plus-game* (*x*, *z*))
have *ge-game* (*y*, *z*)

proof −
{ **fix** *yr*
assume *yr*: *zin yr* (*right-options y*)
from *hyp* **have** ¬ (*ge-game* (*plus-game* (*x*, *z*), *plus-game* (*x*, *yr*)))
by (*auto simp add*: *ge-game-def* [*of plus-game* (*x*,*y*) *plus-game*(*x*,*z*)]
right-options-plus zunion zimage-iff intro: *yr*)
then have ¬ (*ge-game* (*z*, *yr*))
apply (*subst induct-hyp*[**where** *y*=[*x*, *z*, *yr*], *of x z yr*])
apply (*simp-all add*: *yr lprod-3-6*)
done
}
note *yr* = *this*
{ **fix** *zl*
assume *zl*: *zin zl* (*left-options z*)
from *hyp* **have** ¬ (*ge-game* (*plus-game* (*x*, *zl*), *plus-game* (*x*, *y*)))
by (*auto simp add*: *ge-game-def* [*of plus-game* (*x*,*y*) *plus-game*(*x*,*z*)]
left-options-plus zunion zimage-iff intro: *zl*)
then have ¬ (*ge-game* (*zl*, *y*))
apply (*subst goal1* (*1*)[*rule-format*, **where** *y*=[*x*, *zl*, *y*], *of x zl y*])
apply (*simp-all add*: *goal1* (*2*) *zl lprod-3-7*)
done
}
note *zl* = *this*
show *ge-game* (*y*, *z*)
apply (*subst ge-game-def*)
apply (*auto simp add*: *yr zl*)
done
qed
}
note *right-imp-left* = *this*
{
assume *yz*: *ge-game* (*y*, *z*)
{
fix *x'*
assume *x'*: *zin x'* (*right-options x*)
assume *hyp*: *ge-game* (*plus-game* (*x*, *z*), *plus-game* (*x'*, *y*))
then have *n*: ¬ (*ge-game* (*plus-game* (*x'*, *y*), *plus-game* (*x'*, *z*)))
by (*auto simp add*: *ge-game-def* [*of plus-game* (*x*,*z*) *plus-game* (*x'*, *y*)]
right-options-plus zunion zimage-iff intro: *x'*)
have *t*: *ge-game* (*plus-game* (*x'*, *y*), *plus-game* (*x'*, *z*))
apply (*subst induct-hyp*[*symmetric*])
apply (*auto intro*: *lprod-3-3 x' yz*)
done
from *n t* **have** *False* **by** *blast*
}
note *case1* = *this*
{
fix *x'*
assume *x'*: *zin x'* (*left-options x*)

```
        assume hyp: ge-game (plus-game (x′, z), plus-game (x, y))
        then have n: ¬ (ge-game (plus-game (x′, y), plus-game (x′, z)))
          by (auto simp add: ge-game-def[of plus-game (x′,z) plus-game (x, y)]
            left-options-plus zunion zimage-iff intro: x′)
        have t: ge-game (plus-game (x′, y), plus-game (x′, z))
          apply (subst induct-hyp[symmetric])
          apply (auto intro: lprod-3-3 x′ yz)
          done
        from n t have False by blast
      }
      note case3 = this
      {
        fix y′
        assume y′: zin y′ (right-options y)
        assume hyp: ge-game (plus-game(x, z), plus-game (x, y′))
        then have ge-game(z, y′)
          apply (subst induct-hyp[of [x, z, y′] x z y′])
          apply (auto simp add: hyp lprod-3-6 y′)
          done
        with yz have ge-game (y, y′)
          by (blast intro: ge-game-trans)
        with y′ have False by (auto simp add: ge-game-leftright-refl)
      }
      note case2 = this
      {
        fix z′
        assume z′: zin z′ (left-options z)
        assume hyp: ge-game (plus-game(x, z′), plus-game (x, y))
        then have ge-game(z′, y)
          apply (subst induct-hyp[of [x, z′, y] x z′ y])
          apply (auto simp add: hyp lprod-3-7 z′)
          done
        with yz have ge-game (z′, z)
          by (blast intro: ge-game-trans)
        with z′ have False by (auto simp add: ge-game-leftright-refl)
      }
      note case4 = this
      have ge-game(plus-game (x, y), plus-game (x, z))
        apply (subst ge-game-def)
      apply (auto simp add: right-options-plus left-options-plus zunion zimage-iff)
        apply (auto intro: case1 case2 case3 case4)
        done
    }
    note left-imp-right = this
    show ?case by (auto intro: right-imp-left left-imp-right)
  qed
}
note a = this[of [x, y, z]]
then show ?thesis by blast
```

qed

lemma *ge-plus-game-right*: *ge-game* (y,z) = *ge-game*(*plus-game* (y, x), *plus-game* (z, x))
  by (*simp add*: *ge-plus-game-left plus-game-comm*)

lemma *ge-neg-game*: *ge-game* (*neg-game* x, *neg-game* y) = *ge-game* (y, x)
proof −
  { fix *a*
    have $\forall$ x y. $a = [x, y] \longrightarrow$ *ge-game* (*neg-game* x, *neg-game* y) = *ge-game* (y, x)
    proof (*induct a rule*: *induct-game*, (*rule impI* | *rule allI*)+)
      case (*goal1 a x y*)
      note *ihyp* = *goal1*(*1*)[*rule-format*, *simplified goal1*(*2*)]
      { fix *xl*
        assume *xl*: *zin xl* (*left-options x*)
        have *ge-game* (*neg-game y*, *neg-game xl*) = *ge-game* (*xl*, *y*)
          apply (*subst ihyp*)
          apply (*auto simp add*: *lprod-2-1 xl*)
          done
      }
      note *xl* = *this*
      { fix *yr*
        assume *yr*: *zin yr* (*right-options y*)
        have *ge-game* (*neg-game yr*, *neg-game x*) = *ge-game* (*x*, *yr*)
          apply (*subst ihyp*)
          apply (*auto simp add*: *lprod-2-2 yr*)
          done
      }
      note *yr* = *this*
      show *?case*
        by (*auto simp add*: *ge-game-def*[*of neg-game x neg-game y*] *ge-game-def*[*of y x*]
          right-options-neg left-options-neg zimage-iff xl yr)
    qed
  }
  note *a* = *this*[*of* $[x,y]$]
  then show *?thesis* **by** *blast*
qed

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

typedef *Pg* = *UNIV*//*eq-game-rel*
  by (*auto simp add*: *quotient-def*)

lemma *equiv-eq-game*[*simp*]: *equiv UNIV eq-game-rel*
  by (*auto simp add*: *equiv-def refl-def sym-def trans-def eq-game-rel-def*

eq-game-sym intro: *eq-game-refl eq-game-trans*)

instance *Pg* :: {*ord*,*zero*,*plus*,*minus*} **..**

defs (**overloaded**)
Pg-zero-def: $0 \equiv$ *Abs-Pg* (*eq-game-rel* `` {*zero-game*})
Pg-le-def: $G \le H \equiv \exists\ g\ h.\ g \in$ *Rep-Pg* $G \wedge h \in$ *Rep-Pg* $H \wedge$ *ge-game* (h, g)
Pg-less-def: $G < H \equiv G \le H \wedge G \neq (H{::}Pg)$
Pg-minus-def: $-\ G \equiv$ *contents* ($\bigcup\ g \in$ *Rep-Pg* G. {*Abs-Pg* (*eq-game-rel* `` {*neg-game* g}})})
Pg-plus-def: $G + H \equiv$ *contents* ($\bigcup\ g \in$ *Rep-Pg* G. $\bigcup\ h \in$ *Rep-Pg* H. {*Abs-Pg* (*eq-game-rel* `` {*plus-game* (g,h)}})})
Pg-diff-def: $G - H \equiv G + (-\ (H{::}Pg))$

lemma *Rep-Abs-eq-Pg*[*simp*]: *Rep-Pg* (*Abs-Pg* (*eq-game-rel* `` {$g$})) = *eq-game-rel* `` {g}
apply (*subst Abs-Pg-inverse*)
apply (*auto simp add*: *Pg-def quotient-def*)
done

lemma *char-Pg-le*[*simp*]: (*Abs-Pg* (*eq-game-rel* `` {$g$}) $\le$ *Abs-Pg* (*eq-game-rel* `` {h})) = (*ge-game* (h, g))
apply (*simp add*: *Pg-le-def*)
apply (*auto simp add*: *eq-game-rel-def eq-game-def intro*: *ge-game-trans ge-game-refl*)
done

lemma *char-Pg-eq*[*simp*]: (*Abs-Pg* (*eq-game-rel* `` {$g$}) = *Abs-Pg* (*eq-game-rel* `` {h})) = (*eq-game g h*)
apply (*simp add*: *Rep-Pg-inject* [*symmetric*])
apply (*subst eq-equiv-class-iff* [*of UNIV*])
apply (*simp-all*)
apply (*simp add*: *eq-game-rel-def*)
done

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 −
have (λ *g h*. {*Abs-Pg* (*eq-game-rel* `` {*plus-game* (g, h)})}) *respects2 eq-game-rel*

apply (*simp add*: *congruent2-def*)
apply (*auto simp add*: *eq-game-rel-def eq-game-def*)
apply (*rule-tac y=plus-game* (*y1*, *z2*) **in** *ge-game-trans*)
apply (*simp add*: *ge-plus-game-left*[*symmetric*] *ge-plus-game-right*[*symmetric*])+
apply (*rule-tac y=plus-game* (*z1*, *y2*) **in** *ge-game-trans*)
apply (*simp add*: *ge-plus-game-left*[*symmetric*] *ge-plus-game-right*[*symmetric*])+
done
then show *?thesis*
by (*simp add*: *Pg-plus-def UN-equiv-class2*[*OF equiv-eq-game equiv-eq-game*])
qed

lemma *char-Pg-minus*[*simp*]: $-$ *Abs-Pg* (*eq-game-rel* `` {*g*}) = *Abs-Pg* (*eq-game-rel* `` {*neg-game g*})
proof $-$
  have (λ *g*. {*Abs-Pg* (*eq-game-rel* `` {*neg-game g*})}) *respects eq-game-rel*
    apply (*simp add*: *congruent-def*)
    apply (*auto simp add*: *eq-game-rel-def eq-game-def ge-neg-game*)
    done
  then show *?thesis*
    by (*simp add*: *Pg-minus-def UN-equiv-class*[*OF equiv-eq-game*])
qed

lemma *eq-Abs-Pg*[*rule-format*, *cases type*: *Pg*]: ($\forall$ *g*. *z* = *Abs-Pg* (*eq-game-rel* `` {*g*}) $\longrightarrow$ *P*) $\longrightarrow$ *P*
  apply (*cases z*, *simp*)
  apply (*simp add*: *Rep-Pg-inject*[*symmetric*])
  apply (*subst Abs-Pg-inverse*, *simp*)
  apply (*auto simp add*: *Pg-def quotient-def*)
  done

instance *Pg* :: *pordered-ab-group-add*
proof
  fix *a b c* :: *Pg*
  show $(a < b) = (a \leq b \wedge a \neq b)$ **by** (*simp add*: *Pg-less-def*)
  show $a - b = a + (- b)$ **by** (*simp add*: *Pg-diff-def*)
  {
    assume *ab*: $a \leq b$
    assume *ba*: $b \leq a$
    from *ab ba* **show** $a = b$
      apply (*cases a*, *cases b*)
      apply (*simp add*: *eq-game-def*)
      done
  }
  show $a + b = b + a$
    apply (*cases a*, *cases b*)
    apply (*simp add*: *eq-game-def plus-game-comm*)
    done
  show $a + b + c = a + (b + c)$
    apply (*cases a*, *cases b*, *cases c*)
    apply (*simp add*: *eq-game-def plus-game-assoc*)
    done
  show $0 + a = a$
    apply (*cases a*)
    apply (*simp add*: *Pg-zero-def plus-game-zero-left*)
    done
  show $- a + a = 0$
    apply (*cases a*)
    apply (*simp add*: *Pg-zero-def eq-game-plus-inverse plus-game-comm*)
    done

```
  show a ≤ a
    apply (cases a)
    apply (simp add: ge-game-refl)
    done
  {
    assume ab: a ≤ b
    assume bc: b ≤ c
    from ab bc show a ≤ c
      apply (cases a, cases b, cases c)
      apply (auto intro: ge-game-trans)
      done
  }
  {
    assume ab: a ≤ b
    from ab show c + a ≤ c + b
      apply (cases a, cases b, cases c)
      apply (simp add: ge-plus-game-left[symmetric])
      done
  }
qed

end
```