ZF

Steven Obua

April 19, 2009

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* ⇒ *ZF* **and**
Power :: *ZF* ⇒ *ZF* **and**
Repl :: *ZF* ⇒ (*ZF* ⇒ *ZF*) ⇒ *ZF* **and**
Inf :: *ZF*

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

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

constdefs
Sep:: *ZF* ⇒ (*ZF* ⇒ *bool*) ⇒ *ZF*
Sep A p == (*if* (!*x*. *Elem x A* ⟶ *Not* (*p x*)) *then Empty else*
(*let z* = (ϵ *x*. *Elem x A* & *p x*) *in*
let f = % *x*. (*if p x then x else z*) *in Repl A f*))

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

theorem *Sep*: *Elem b* (*Sep A p*) = (*Elem b A* & *p b*)
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* | *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*) ⟹ *Elem f* (*PFun U V*)
by (*simp add*: *Fun-def Sep*)

lemma *Elem-Elem-Fun*: *Elem F* (*Fun U V*) ⟹ *Elem p F* ⟹ *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*) ⟹ *Elem x F* ⟹ *Elem y F* ⟹ *Fst x* = *Fst y* ⟹ *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*⟧ ⟹ ∃ *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*)⟧ ⟹ *?*! *y*. *Elem* (*Opair x y*) *f*
by (*auto simp add*: *Domain isFun-def*)

lemma *fun-value-in-range*: ⟦*isFun f*; *Elem x* (*Domain f*)⟧ ⟹ *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*)⟧ ⟹ *? 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*) ⟹ *? 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 *notsym-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: notsym-Elem[of a a, simplified])

lemma antisym-Elem: Elem a b ⟹ Not (Elem b a)
  apply (insert notsym-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* $\Rightarrow$ *ZF* $\Rightarrow$ *ZF*
Ext-ZF R s $==$ *implode* (*Ext R s*)

lemma *Elem-implode*: $A \in$ *range explode* $\Longrightarrow$ *Elem x* (*implode A*) $= (x \in A)$
apply (*auto*)
apply (*simp-all add*: *explode-def*)
done

lemma *Elem-Ext-ZF*: *set-like R* $\Longrightarrow$ *Elem x* (*Ext-ZF R s*) $= ((x,s) \in R)$
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* $\Rightarrow$ *ZF* $\Rightarrow$ *nat* $\Rightarrow$ *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* $\Rightarrow$ *ZF* $\Rightarrow$ *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*: $(y, x) \in R$
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 ∈ ?C ∧ (∀ y. (y, x) ∈ R ⟶ y ∉ ?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 (∀ x. (∀ y. (y, x) ∈ R ⟶ P y) ⟶ P x) ⟶ (! 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 ⟹ x ∈ (Ext (R^+) s) ⟹ 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 ⟹ 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 ∈ (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 ⟹ ! x. Elem x (Ext-ZF-n R s n) ⟶ x ∈ (Ext (R^+) s)
 apply (induct-tac n)
 apply (auto simp add: Elem-Ext-ZF Ext-def Sum Repl)
 done

lemma set-like R ⟹ 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* ⟹ *regular R*
proof (*simp add*: *regular-def*, *rule allI*)
assume *wf*: *wf R*
fix *A*
show *A* ≠ *Empty* ⟶ (∃ *x*. *Elem x A* ∧ (∀ *y*. (*y*, *x*) ∈ *R* ⟶ ¬ *Elem y A*))
proof
assume *A*: *A* ≠ *Empty*
then have *?* *x*. *x* ∈ *explode A*
by (*auto simp add*: *explode-def Ext Empty*)
then obtain *x* **where** *x*:*x* ∈ *explode A* **..**
from *iffD1*[*OF wf-eq-minimal wf*, *rule-format*, **where** *Q*=*explode A*, *OF x*]
obtain *z* **where** *z* ∈ *explode A* ∧ (∀ *y*. (*y*, *z*) ∈ *R* ⟶ *y* ∉ *explode A*) **by** *auto*

then show ∃ *x*. *Elem x A* ∧ (∀ *y*. (*y*, *x*) ∈ *R* ⟶ ¬ *Elem y A*)
apply (*rule-tac exI*[**where** *x* = *z*])
apply (*simp add*: *explode-def*)
done
qed
qed

lemma *wf-eq-wfzf*: (*wf R* ∧ *set-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* ⟹ *wfzf* (*R*^+)
by (*auto simp add*: *wf-eq-wfzf*[*symmetric*] *implodeable-Ext-trancl wf-trancl*)

lemma *Ext-subset-mono*: *R* ⊆ *S* ⟹ *Ext R y* ⊆ *Ext S y*
by (*auto simp add*: *Ext-def*)

lemma *set-like-subset*: *set-like R* ⟹ *S* ⊆ *R* ⟹ *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* ∈ (*Ext S y*)) **in** *exI*)
apply (*auto simp add*: *explode-def Sep*)
done

lemma *wfzf-subset*: *wfzf S* ⟹ *R* ⊆ *S* ⟹ *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* ∈ *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* ⇒ *'a zet* ⇒ *'a zet*
zunion a b ≡ *Abs-zet* ((*Rep-zet a*) ∪ (*Rep-zet b*))
zsubset :: *'a zet* ⇒ *'a zet* ⇒ *bool*
zsubset a b ≡ ! *x*. *zin x a* ⟶ *zin x b*

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

lemma *Rep-zet-zunion*: *Rep-zet* (*zunion a b*) = (*Rep-zet a*) ∪ (*Rep-zet b*)
proof −
from *Rep-zet*[*of a*] **have** *? f z*. *inj-on f* (*Rep-zet a*) ∧ *f* ‘ (*Rep-zet a*) = *explode z*
by (*auto simp add*: *zet-def'*)
then obtain *fa za* **where** *a*:*inj-on fa* (*Rep-zet a*) ∧ *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*) ∧ *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 *List 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**
[*code del*]: {#} = *Abs-multiset* (λ*a*. *0*)

definition *single* :: *'a* => *'a multiset* **where**
[*code del*]: *single 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* (λ*x*. *if P x then Rep-multiset M x else 0*)

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

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

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

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

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

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

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

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

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

instance ..

end

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

Multiset Enumeration

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

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: $(\lambda a.\ 0) \in multiset$
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset*: $(\lambda b.\ if\ b = a\ then\ 1\ else\ 0) \in multiset$
by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset*:
$M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
by (*simp add*: *multiset-def*)

lemma *diff-preserves-multiset*:
$M \in multiset \Longrightarrow (\lambda a.\ M\ a - N\ a) \in multiset$
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*)
apply *auto*
done

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

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

1.2 Algebraic properties

1.2.1 Union

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

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

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

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* ∧ {#} − *M* = {#}
by (*simp add*: *Mempty-def diff-def in-multiset*)

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

lemma *diff-cancel*: *A* − *A* = {#}
by (*simp add*: *diff-def Mempty-def*)

1.2.3 Count of elements

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

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

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

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

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 in-multiset*)

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*: *set-of-def Mempty-def in-multiset count-def expand-fun-eq* [**where** *f*=*Rep-multiset M*])

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

lemma *set-of-MCollect* [*simp*]: *set-of* {# *x*:#*M*. *P x* #} = *set-of M* ∩ {*x*. *P x*}
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 simp*: *in-multiset*)
apply (*simp add*: *set-of-def count-def in-multiset expand-fun-eq*)
done

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
by (*metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty*)

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

apply *auto*
done

1.2.6 Equality of multisets

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

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
by (*simp add*: *single-def Mempty-def in-multiset expand-fun-eq*)

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

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

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

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

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

lemma *union-is-single*:
$(M + N = \{\#a\#\}) = (M = \{\#a\#\} \wedge N=\{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\})$
apply (*simp add*: *Mempty-def single-def union-def in-multiset add-is-1 expand-fun-eq*)
apply *blast*
done

lemma *single-is-union*:
$(\{\#a\#\} = M + N) \longleftrightarrow (\{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N)$
apply (*unfold Mempty-def single-def union-def*)
apply (*simp add*: *add-is-1 one-is-add in-multiset 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*: *in-multiset 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 ⟹ b = c* **by** *simp*
qed

lemma *insert-DiffM*:
  x ∈# M ⟹ {#x#} + (M − {#x#}) = M
by (*clarsimp simp*: *multiset-eq-conv-count-eq*)

lemma *insert-DiffM2*[*simp*]:
  x ∈# M ⟹ M − {#x#} + {#x#} = M
by (*clarsimp simp*: *multiset-eq-conv-count-eq*)

lemma *multi-union-self-other-eq*:
  (*A*::*'a multiset*) *+ X = A + Y ⟹ X = Y*
by (*induct A arbitrary*: *X Y*) *auto*

lemma *multi-self-add-other-not-self*[*simp*]: *(A = A + {#x#}) = False*
by (*metis single-not-empty union-empty union-left-cancel*)

lemma *insert-noteq-member*:
  assumes *BC*: *B + {#b#} = C + {#c#}*
   and *bnotc*: *b ≠ c*
  shows *c ∈# B*
proof −
  have *c ∈# C + {#c#}* **by** *simp*
  have *nc*: *¬ c ∈# {#b#}* **using** *bnotc* **by** *simp*
  then have *c ∈# B + {#b#}* **using** *BC* **by** *simp*
  then show *c ∈# B* **using** *nc* **by** *simp*
qed

lemma *add-eq-conv-ex*:
  (M + {#a#} = N + {#b#}) =
    (M = N ∧ a = b ∨ (∃K. M = K + {#b#} ∧ N = K + {#a#}))
by (*auto simp add*: *add-eq-conv-diff*)

lemma *empty-multiset-count*:
  (∀x. count A x = 0) = (A = {#})
by (*metis count-empty multiset-eq-conv-count-eq*)

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-inter-single*: $a \neq b \Longrightarrow \{\#a\#\} \#\cap \{\#b\#\} = \{\#\}$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*)

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

1.2.8 Comprehension (filter)

lemma *MCollect-empty* [*simp*]: *MCollect* $\{\#\}$ $P = \{\#\}$
by (*simp add*: *MCollect-def Mempty-def Abs-multiset-inject*
in-multiset expand-fun-eq)

lemma *MCollect-single* [*simp*]:
MCollect $\{\#x\#\}$ P = (*if* P x *then* $\{\#x\#\}$ *else* $\{\#\}$)
by (*simp add*: *MCollect-def Mempty-def single-def Abs-multiset-inject*
in-multiset expand-fun-eq)

lemma *MCollect-union* [*simp*]:
MCollect $(M+N)$ f = *MCollect* M f + *MCollect* N f
by (*simp add*: *MCollect-def union-def Abs-multiset-inject*
in-multiset expand-fun-eq)

1.3 Induction and case splits

lemma *setsum-decr*:
finite F ==> (0::nat) < f a ==>
setsum (f (a := f a − 1)) F = (if a∈F then setsum f F − 1 else setsum f F)
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* (λ*a*. (*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* = (λ*a.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 ∈ *multiset ==> P* (λ*a*. *0*) ==>
(!!*f b*. *f* ∈ *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* $f(b := f\ b + 1) = (\lambda a.\ f\ a + (\textit{if } a{=}b \textit{ then } 1 \textit{ else } 0))$)
prefer *2*
apply (*simp add*: *expand-fun-eq*)
apply (*erule ssubst*)
apply (*erule Abs-multiset-inverse* [*THEN subst*])
apply (*drule add* [*unfolded defns*, *simplified*])
apply(*simp add*:*in-multiset*)
done
qed

lemma *multi-nonempty-split*: $M \neq \{\#\} \Longrightarrow \exists A\ a.\ M = A + \{\#a\#\}$
by (*induct M*) *auto*

lemma *multiset-cases* [*cases type*, *case-names empty add*]:
assumes *em*: $M = \{\#\} \Longrightarrow P$
assumes *add*: $\bigwedge N\ x.\ M = N + \{\#x\#\} \Longrightarrow P$
shows P
proof (*cases M* = {#})
assume $M = \{\#\}$ **then show** *?thesis* **using** *em* **by** *simp*
next
assume $M \neq \{\#\}$
then obtain $M'\ m$ **where** $M = M' + \{\#m\#\}$
by (*blast dest*: *multi-nonempty-split*)
then show *?thesis* **using** *add* **by** *simp*
qed

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = A + \{\#x\#\}$
apply (*cases M*)
apply *simp*
apply (*rule-tac x*=$M - \{\#x\#\}$ **in** *exI*, *simp*)
done

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

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

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
by (*cases* $B = \{\#\}$) (*auto dest*: *multi-member-split*)

1.4 Orderings

1.4.1 Well-foundedness

definition *mult1* :: $('a \times 'a)$ *set* => $('a$ *multiset* $\times$ $'a$ *multiset*$)$ *set* **where**
[*code del*]: *mult1* $r = \{(N, M).\ \exists a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$
$(\forall b.\ b :\# K \longrightarrow (b, a) \in r)\}$

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

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
by (*simp add*: *mult1-def*)

lemma *less-add*: $(N, M0 + \{\#a\#\}) \in mult1\ r$ ==>
$(\exists M.\ (M, M0) \in mult1\ r \wedge N = M + \{\#a\#\}) \vee$
$(\exists K.\ (\forall b.\ b :\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
(**is** - $\Longrightarrow$ *?case1* $(mult1\ r) \vee$ *?case2*)
proof (*unfold mult1-def*)
let *?r* $= \lambda K\ a.\ \forall b.\ b :\# K \longrightarrow (b, a) \in r$
let *?R* $= \lambda N\ M.\ \exists a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$ *?r* $K\ a$
let *?case1* = *?case1* $\{(N, M).$ *?R* $N\ M\}$

assume $(N, M0 + \{\#a\#\}) \in \{(N, M).$ *?R* $N\ M\}$
then have $\exists a'\ M0'\ K.$
$M0 + \{\#a\#\} = M0' + \{\#a'\#\} \wedge N = M0' + K \wedge$ *?r* $K\ a'$ **by** *simp*
then show *?case1* $\vee$ *?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 $M0 = M0' \wedge a = a' \vee$
$(\exists K'.\ M0 = K' + \{\#a'\#\} \wedge M0' = K' + \{\#a\#\})$
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* $K\ a \wedge N = M0 + K$ **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* = $\lambda M.\ M - \{\#a\#\}$ **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* ==> ∃ *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 ==>
∀ *I J K*. (*size J* = *n* ∧ *J* ≠ {#} ∧ (∀ *k* ∈ *set-of K*. ∃ *j* ∈ *set-of J*. (*k*, *j*) ∈ *r*))
−−> (*I* + *K*, *I* + *J*) ∈ *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'* ≠ {#}.

apply (*cut-tac M* = *K* **and** *P* = $\lambda x.\ (x,\ a) \in r$ **in** *multiset-partition*)
apply (*erule-tac P* = ∀ *k* ∈ *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*) ∈ *r* #}) + {# *x* :# *K*. (*x*, *a*) ∉ *r* #},
(*I* + {# *x* :# *K*. (*x*, *a*) ∈ *r* #}) + *J'*) ∈ *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*:
$\mathit{trans}\ r \Longrightarrow J \neq \{\#\} \Longrightarrow \forall k \in \mathit{set\text{-}of}\ K.\ \exists j \in \mathit{set\text{-}of}\ J.\ (k, j) \in r \Longrightarrow (I + K, I + J) \in \mathit{mult}\ r$
using *one-step-implies-mult-aux* **by** *blast*

1.4.3 Partial-order properties

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

definition *less-multiset-def* [*code del*]:
$M' < M \longleftrightarrow (M', M) \in \mathit{mult}\ \{(x', x).\ x' < x\}$

definition *le-multiset-def* [*code del*]:
$M' <= M \longleftrightarrow M' = M \vee M' < (M::'a\ \mathit{multiset})$

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

lemma *mult-less-not-refl*: $\neg\ M < (M::'a::order\ \mathit{multiset})$
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!*]: $M < (M::'a::order\ \mathit{multiset}) \Longrightarrow R$
using *insert mult-less-not-refl* **by** *fast*

Transitivity.

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

Asymmetry.

theorem *mult-less-not-sym*: $M < N \Longrightarrow \neg\ N < (M::'a::order\ multiset)$
apply *auto*
apply (*rule mult-less-not-refl* [*THEN notE*])
apply (*erule mult-less-trans*, *assumption*)
done

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

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

Anti-symmetry.

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

Transitivity.

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

instance proof
qed (*auto simp add*: *mult-less-le dest*: *mult-le-antisym elim*: *mult-le-trans*)

end

1.4.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in mult1\ r \Longrightarrow trans\ r \Longrightarrow (C + B, C + D) \in mult1\ r$
apply (*unfold mult1-def*)
apply *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*: $B < D \Longrightarrow C + B < C + (D::'a::order\ multiset)$
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*: $B < D \Longrightarrow B + C < D + (C::'a::order\ multiset)$
apply (*subst union-commute* [*of B C*])
apply (*subst union-commute* [*of D C*])
apply (*erule union-less-mono2*)
done

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

lemma *union-le-mono*:
$A <= C \Longrightarrow B <= D \Longrightarrow A + B <= C + (D::'a::order\ multiset)$
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*)
apply *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

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

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

lemma *multiset-of-zero-iff-right*[*simp*]: $(\{\#\} = \textit{multiset-of}\ x) = (x = [])$
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*: $x \in \textit{set}\ xs = (x :\# \textit{multiset-of}\ xs)$
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*)
apply (*rule allI*)
apply (*rule-tac M* = *y* **in** *multiset-induct*)
apply *auto*
apply (*rule-tac x* = *x* # *xa* **in** *exI*)
apply *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* ∈ *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* ⟹ *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* ⟹
(*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-of [x←xs. P x] + multiset-of [x←xs. ¬P x] = multiset-of xs
by (induct xs) (auto simp: union-ac)

lemma count-filter:
  count (multiset-of xs) x = length [y ← xs. y = x]
by (induct xs) auto

lemma nth-mem-multiset-of: i < length ls ⟹ (ls ! i) :# multiset-of ls
apply (induct ls arbitrary: i)
 apply simp
apply (case-tac i)
 apply auto
done

lemma multiset-of-remove1: multiset-of (remove1 a xs) = multiset-of xs − {#a#}
by (induct xs) (auto simp add: multiset-eq-conv-count-eq)

lemma multiset-of-eq-length:
assumes multiset-of xs = multiset-of ys
shows length xs = length ys
using assms
proof (induct arbitrary: ys rule: length-induct)
  case (1 xs ys)
  show ?case
  proof (cases xs)
    case Nil with 1.prems show ?thesis by simp
  next
    case (Cons x xs')
    note xCons = Cons
    show ?thesis
    proof (cases ys)
      case Nil
      with 1.prems Cons show ?thesis by simp
    next
      case (Cons y ys')
      have x-in-ys: x = y ∨ x ∈ set ys'
      proof (cases x = y)
        case True then show ?thesis ..
      next
        case False
       from 1.prems [symmetric] xCons Cons have x :# multiset-of ys' + {#y#}
by simp
        with False show ?thesis by (simp add: mem-set-multiset-eq)
      qed
```

```
    from 1.hyps have IH: length xs' < length xs ⟶
      (∀ x. multiset-of xs' = multiset-of x ⟶ length xs' = length x) by blast
   from 1.prems x-in-ys Cons xCons have multiset-of xs' = multiset-of (remove1
x (y#ys'))
      apply −
      apply (simp add: multiset-of-remove1, simp only: add-eq-conv-diff)
      apply fastsimp
      done
   with IH xCons have IH': length xs' = length (remove1 x (y#ys')) by fastsimp
    from x-in-ys have x ≠ y ⟹ length ys' > 0 by auto
   with Cons xCons x-in-ys IH' show ?thesis by (auto simp add: length-remove1)
   qed
 qed
qed
```

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

```
lemma properties-for-sort:
  multiset-of ys = multiset-of xs ⟹ sorted ys ⟹ sort xs = ys
proof (induct xs arbitrary: ys)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  then have x ∈ set ys
    by (auto simp add:  mem-set-multiset-eq intro!: ccontr)
  with Cons.prems Cons.hyps [of remove1 x ys] show ?case
    by (simp add: sorted-remove1 multiset-of-remove1 insort-remove1)
qed
```

1.6 Pointwise ordering induced by count

```
definition mset-le :: 'a multiset ⇒ 'a multiset ⇒ bool  (infix ≤# 50) where
  [code del]: A ≤# B ⟷ (∀ a. count A a ≤ count B a)

definition mset-less :: 'a multiset ⇒ 'a multiset ⇒ bool  (infix <# 50) where
  [code del]: A <# B ⟷ A ≤# B ∧ A ≠ B

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

lemma mset-le-refl[simp]: A ≤# A
unfolding mset-le-def by auto

lemma mset-le-trans: A ≤# B ⟹ B ≤# C ⟹ A ≤# C
unfolding mset-le-def by (fast intro: order-trans)

lemma mset-le-antisym: A ≤# B ⟹ B ≤# A ⟹ 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*]: $A \leq\# A + B$
unfolding *mset-le-def* **by** *auto*

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

lemma *mset-le-single*: $a :\# B \Longrightarrow \{\#a\#\} \leq\# B$
by (*simp add*: *mset-le-def*)

lemma *multiset-diff-union-assoc*: $C \leq\# B \Longrightarrow A + B - C = A + (B - C)$
by (*simp add*: *multiset-eq-conv-count-eq mset-le-def*)

lemma *mset-le-multiset-union-diff-commute*:
assumes $B \leq\# A$
shows $A - B + C = A + C - B$
proof −
  from *mset-le-exists-conv* [*of B A*] *assms* **have** $\exists D.\ A = B + D$ **..**
  from *this* **obtain** D **where** $A = B + D$ **..**
  then show *?thesis*
    apply *simp*
    apply (*subst union-commute*)
    apply (*subst multiset-diff-union-assoc*)
    apply *simp*
    apply (*simp add*: *diff-cancel*)
    apply (*subst union-assoc*)
    apply (*subst union-commute*[*of B* -])
    apply (*subst multiset-diff-union-assoc*)
    apply *simp*

```
    apply (simp add: diff-cancel)
    done
qed

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

lemma multiset-of-update:
  i < length ls ⟹ multiset-of (ls[i := v]) = multiset-of ls − {#ls ! i#} + {#v#}
proof (induct ls arbitrary: i)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0 then show ?thesis by simp
  next
    case (Suc i')
    with Cons show ?thesis
      apply simp
      apply (subst union-assoc)
      apply (subst union-commute [where M = {#v#} and N = {#x#}])
      apply (subst union-assoc [symmetric])
      apply simp
      apply (rule mset-le-multiset-union-diff-commute)
      apply (simp add: mset-le-single nth-mem-multiset-of)
      done
  qed
qed

lemma multiset-of-swap:
  i < length ls ⟹ j < length ls ⟹
    multiset-of (ls[j := ls ! i, i := ls ! j]) = multiset-of ls
apply (case-tac i = j)
 apply simp
apply (simp add: multiset-of-update)
apply (subst elem-imp-eq-diff-union[symmetric])
 apply (simp add: nth-mem-multiset-of)
apply simp
done

interpretation mset-order: order op ≤# op <#
proof qed (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 + op ≤# op <#
proof qed (*erule mset-le-mono-add* [*OF mset-le-refl*])

interpretation *mset-order-semigroup-cancel*:
  pordered-ab-semigroup-add-imp-le op + op ≤# op <#
proof qed *simp*

lemma *mset-lessD*: $A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x=x* **in** *allE*)
apply *auto*
done

lemma *mset-leD*: $A \subseteq\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x = x* **in** *allE*)
apply *auto*
done

lemma *mset-less-insertD*: $(A + \{\#x\#\} \subset\# B) \Longrightarrow (x \in\# B \wedge A \subset\# B)$
apply (*rule conjI*)
 apply (*simp add*: *mset-lessD*)
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply *safe*
 apply (*erule-tac x = a* **in** *allE*)
 apply (*auto split*: *split-if-asm*)
done

lemma *mset-le-insertD*: $(A + \{\#x\#\} \subseteq\# B) \Longrightarrow (x \in\# B \wedge A \subseteq\# B)$
apply (*rule conjI*)
 apply (*simp add*: *mset-leD*)
apply (*force simp*: *mset-le-def mset-less-def split*: *split-if-asm*)
done

lemma *mset-less-of-empty*[*simp*]: $A \subset\# \{\#\} = False$
by (*induct A*) (*auto simp*: *mset-le-def mset-less-def*)

lemma *multi-psub-of-add-self*[*simp*]: $A \subset\# A + \{\#x\#\}$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *multi-psub-self*[*simp*]: $A \subset\# A = False$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-add-bothsides*:
  $T + \{\#x\#\} \subset\# S + \{\#x\#\} \Longrightarrow T \subset\# S$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-empty-nonempty*: $(\{\#\} \subset\# S) = (S \neq \{\#\})$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-size*: $A \subset\# B \Longrightarrow$ *size* $A <$ *size* B
proof (*induct A arbitrary*: B)
  case (*empty M*)
  then have $M \neq \{\#\}$ **by** (*simp add*: *mset-less-empty-nonempty*)
  then obtain M' x **where** $M = M' + \{\#x\#\}$
    by (*blast dest*: *multi-nonempty-split*)
  then show *?case* **by** *simp*
next
  case (*add S x T*)
  have *IH*: $\bigwedge B.\ S \subset\# B \Longrightarrow$ *size* $S <$ *size* B **by** *fact*
  have *SxsubT*: $S + \{\#x\#\} \subset\# T$ **by** *fact*
  then have $x \in\# T$ **and** $S \subset\# T$ **by** (*auto dest*: *mset-less-insertD*)
  then obtain T' **where** *T*: $T = T' + \{\#x\#\}$
    by (*blast dest*: *multi-member-split*)
  then have $S \subset\# T'$ **using** *SxsubT*
    by (*blast intro*: *mset-less-add-bothsides*)
  then have *size* $S <$ *size* T' **using** *IH* **by** *simp*
  then show *?case* **using** T **by** *simp*
qed

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

lemma *mset-less-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
by (*auto simp*: *mset-le-def mset-less-def multi-drop-mem-not-eq*)

1.7 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

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

lemma *multiset-add-sub-el-shuffle*:
  assumes $c \in\# B$ **and** $b \neq c$
  shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
proof −
  from ‹$c \in\# B$› **obtain** A **where** *B*: $B = A + \{\#c\#\}$
    by (*blast dest*: *multi-member-split*)
  have $A + \{\#b\#\} = A + \{\#b\#\} + \{\#c\#\} - \{\#c\#\}$ **by** *simp*
  then have $A + \{\#b\#\} = A + \{\#c\#\} + \{\#b\#\} - \{\#c\#\}$
    by (*simp add*: *union-ac*)
  then show *?thesis* **using** B **by** *simp*
qed

lemma *wf-mset-less-rel*: *wf mset-less-rel*
apply (*unfold mset-less-rel-def*)
apply (*rule wf-measure* [*THEN wf-subset*, **where** *f1*=*size*])
apply (*clarsimp simp*: *measure-def inv-image-def mset-less-size*)
done

The induction rules:

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall A.\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows *P B*
apply (*rule wf-mset-less-rel* [*THEN wf-induct*])
apply (*rule ih*, *auto simp*: *mset-less-rel-def*)
done

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (F + \{\#a\#\})$
shows *P F*
proof −
from ‹$F \subseteq\# A$›
show *?thesis*
proof (*induct F*)
show $P\ \{\#\}$ **by** *fact*
next
fix *x F*
assume *P*: $F \subseteq\# A \Longrightarrow P\ F$ **and** *i*: $F + \{\#x\#\} \subseteq\# A$
show $P\ (F + \{\#x\#\})$
proof (*rule insert*)
from *i* **show** $x \in\# A$ **by** (*auto dest*: *mset-le-insertD*)
from *i* **have** $F \subseteq\# A$ **by** (*auto dest*: *mset-le-insertD*)
with *P* **show** *P F* **.**
qed
qed
qed

A consequence: Extensionality.

lemma *multi-count-eq*: $(\forall x.\ count\ A\ x = count\ B\ x) = (A = B)$
apply (*rule iffI*)
prefer *2*
apply *clarsimp*
apply (*induct A arbitrary*: *B rule*: *full-multiset-induct*)
apply (*rename-tac C*)
apply (*case-tac B rule*: *multiset-cases*)
apply (*simp add*: *empty-multiset-count*)
apply *simp*
apply (*case-tac x* $\in\# C$)
apply (*force dest*: *multi-member-split*)
apply (*erule-tac x* = *x* **in** *allE*)

apply *simp*
done

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

1.8 The fold combinator

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

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

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

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

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

lemma *Diff1-fold-msetG*:
  fold-msetG f z $(A - \{\#x\#\})$ *y* $\Longrightarrow$ $x \in\# A \Longrightarrow$ *fold-msetG f z A* $(f\ x\ y)$
apply (*frule-tac* $x = x$ **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-msetG-nonempty*: $\exists x.$ *fold-msetG f z A x*
apply (*induct A*)
 apply *blast*
apply *clarsimp*
apply (*drule-tac* $x = x$ **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-mset-empty*[*simp*]: *fold-mset f z* $\{\#\} = z$
unfolding *fold-mset-def* **by** *blast*

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

lemma *fold-msetG-determ*:
fold-msetG f z A x $\Longrightarrow$ *fold-msetG f z A y* $\Longrightarrow$ $y = x$
proof (*induct arbitrary*: *x y z rule*: *full-multiset-induct*)
case (*less M* x_1 x_2 *Z*)
have *IH*: $\forall A.\ A \subset\# M \longrightarrow$
$(\forall x\ x'\ x''.$ *fold-msetG f x'' A x* $\longrightarrow$ *fold-msetG f x'' A x'*
$\longrightarrow x' = x)$ **by** *fact*
have *Mfoldx*$_1$: *fold-msetG f Z M* x_1 **and** *Mfoldx*$_2$: *fold-msetG f Z M* x_2 **by** *fact+*
show *?case*
proof (*rule fold-msetG.cases* [*OF Mfoldx*$_1$])
assume $M = \{\#\}$ **and** $x_1 = Z$
then show *?case* **using** *Mfoldx*$_2$ **by** *auto*
next
fix *B b u*
assume $M = B + \{\#b\#\}$ **and** $x_1 = f\ b\ u$ **and** *Bu*: *fold-msetG f Z B u*
then have *MBb*: $M = B + \{\#b\#\}$ **and** x_1: $x_1 = f\ b\ u$ **by** *auto*
show *?case*
proof (*rule fold-msetG.cases* [*OF Mfoldx*$_2$])
assume $M = \{\#\}$ $x_2 = Z$
then show *?case* **using** *Mfoldx*$_1$ **by** *auto*
next
fix *C c v*
assume $M = C + \{\#c\#\}$ **and** $x_2 = f\ c\ v$ **and** *Cv*: *fold-msetG f Z C v*
then have *MCc*: $M = C + \{\#c\#\}$ **and** x_2: $x_2 = f\ c\ v$ **by** *auto*
then have *CsubM*: $C \subset\# M$ **by** *simp*
from *MBb* **have** *BsubM*: $B \subset\# M$ **by** *simp*
show *?case*
proof *cases*
assume *b=c*
then moreover have $B = C$ **using** *MBb MCc* **by** *auto*
ultimately show *?thesis* **using** *Bu Cv* x_1 x_2 *CsubM IH* **by** *auto*
next
assume *diff*: $b \neq c$
let *?D* $= B - \{\#c\#\}$
have *cinB*: $c \in\# B$ **and** *binC*: $b \in\# C$ **using** *MBb MCc diff*
by (*auto intro*: *insert-noteq-member dest*: *sym*)
have $B - \{\#c\#\} \subset\# B$ **using** *cinB* **by** (*rule mset-less-diff-self*)
then have *DsubM*: *?D* $\subset\#$ *M* **using** *BsubM* **by** (*blast intro*: *mset-less-trans*)
from *MBb MCc* **have** $B + \{\#b\#\} = C + \{\#c\#\}$ **by** *blast*
then have [*simp*]: $B + \{\#b\#\} - \{\#c\#\} = C$
using *MBb MCc binC cinB* **by** *auto*
have *B*: $B =$ *?D* $+ \{\#c\#\}$ **and** *C*: $C =$ *?D* $+ \{\#b\#\}$
using *MBb MCc diff binC cinB*
by (*auto simp*: *multiset-add-sub-el-shuffle*)
then obtain *d* **where** *Dfoldd*: *fold-msetG f Z ?D d*
using *fold-msetG-nonempty* **by** *iprover*
then have *fold-msetG f Z B* (*f c d*) **using** *cinB*
by (*rule Diff1-fold-msetG*)
then have $f\ c\ d = u$ **using** *IH BsubM Bu* **by** *blast*

```
      moreover
      have fold-msetG f Z C (f b d) using binC cinB diff Dfoldd
        by (auto simp: multiset-add-sub-el-shuffle
          dest: fold-msetG.insertI [where x=b])
      then have f b d = v using IH CsubM Cv by blast
      ultimately show ?thesis using x1 x2
        by (auto simp: left-commute)
    qed
  qed
 qed
qed

lemma fold-mset-insert-aux:
  (fold-msetG f z (A + {#x#}) v) =
    (∃ y. fold-msetG f z A y ∧ v = f x y)
apply (rule iffI)
 prefer 2
 apply blast
apply (rule-tac A=A and f=f in fold-msetG-nonempty [THEN exE, standard])
apply (blast intro: fold-msetG-determ)
done

lemma fold-mset-equality: fold-msetG f z A y ⟹ fold-mset f z A = y
unfolding fold-mset-def by (blast intro: fold-msetG-determ)

lemma fold-mset-insert:
  fold-mset f z (A + {#x#}) = f x (fold-mset f z A)
apply (simp add: fold-mset-def fold-mset-insert-aux union-commute)
apply (rule the-equality)
 apply (auto cong add: conj-cong
    simp add: fold-mset-def [symmetric] fold-mset-equality fold-msetG-nonempty)
done

lemma fold-mset-insert-idem:
  fold-mset f z (A + {#a#}) = f a (fold-mset f z A)
apply (simp add: fold-mset-def fold-mset-insert-aux)
apply (rule the-equality)
 apply (auto cong add: conj-cong
    simp add: fold-mset-def [symmetric] fold-mset-equality fold-msetG-nonempty)
done

lemma fold-mset-commute: f x (fold-mset f z A) = fold-mset f (f x z) A
by (induct A) (auto simp: fold-mset-insert left-commute [of x])

lemma fold-mset-single [simp]: fold-mset f z {#x#} = f x z
using fold-mset-insert [of z {#}] by simp

lemma fold-mset-union [simp]:
  fold-mset f z (A+B) = fold-mset f (fold-mset f z A) B
```

proof (*induct A*)
case *empty* **then show** *?case* **by** *simp*
next
case (*add A x*)
have $A + \{\#x\#\} + B = (A+B) + \{\#x\#\}$ **by**(*simp add*:*union-ac*)
then have *fold-mset f z* $(A + \{\#x\#\} + B) =$ *f x* (*fold-mset f z* $(A + B)$)
by (*simp add*: *fold-mset-insert*)
also have $\ldots =$ *fold-mset f* (*fold-mset f z* $(A + \{\#x\#\}))$ *B*
by (*simp add*: *fold-mset-commute*[*of x*,*symmetric*] *add fold-mset-insert*)
finally show *?case* **.**
qed

lemma *fold-mset-fusion*:
assumes *left-commutative g*
shows $(\bigwedge x\ y.\ h\ (g\ x\ y) = f\ x\ (h\ y)) \Longrightarrow h$ (*fold-mset g w A*) = *fold-mset f* (*h w*) *A* (**is** *PROP ?P*)
proof −
interpret *left-commutative g* **by** *fact*
show *PROP ?P* **by** (*induct A*) *auto*
qed

lemma *fold-mset-rec*:
assumes $a \in\# A$
shows *fold-mset f z A* = *f a* (*fold-mset f z* $(A - \{\#a\#\})$)
proof −
from *assms* **obtain** A' **where** $A = A' + \{\#a\#\}$
by (*blast dest*: *multi-member-split*)
then show *?thesis* **by** *simp*
qed

end

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

1.9 Image

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

interpretation *image-left-comm*: *left-commutative op* + *o single o f*
proof qed (*simp add*:*union-ac*)

lemma *image-mset-empty* [*simp*]: *image-mset f* $\{\#\} = \{\#\}$
by (*simp add*: *image-mset-def*)

lemma *image-mset-single* [*simp*]: *image-mset f* {#*x*#} = {#*f x*#}
by (*simp add*: *image-mset-def*)

lemma *image-mset-insert*:
image-mset f (*M* + {#*a*#}) = *image-mset f M* + {#*f a*#}
by (*simp add*: *image-mset-def add-ac*)

lemma *image-mset-union* [*simp*]:
image-mset f (*M*+*N*) = *image-mset f M* + *image-mset f N*
apply (*induct N*)
apply *simp*
apply (*simp add*: *union-assoc* [*symmetric*] *image-mset-insert*)
done

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
by (*induct M*) *simp-all*

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
by (*cases M*) *auto*

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

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

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

1.10 Termination proofs with multiset orders

lemma *multi-member-skip*: *x* ∈# *XS* ⟹ *x* ∈# {# *y* #} + *XS*
and *multi-member-this*: *x* ∈# {# *x* #} + *XS*
and *multi-member-last*: *x* ∈# {# *x* #}
by *auto*

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

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
unfolding *reduction-pair-def ms-strict-def ms-weak-def pair-less-def*

by (*auto intro*: *wf-mult1 wf-trancl simp*: *mult-def*)

lemma *smsI*:
(*set-of A*, *set-of B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
unfolding *ms-strict-def*
by (*rule one-step-implies-mult*) (*auto simp add*: *max-strict-def pair-less-def elim*!:*max-ext.cases*)

lemma *wmsI*:
(*set-of A*, *set-of B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
unfolding *ms-weak-def ms-strict-def*
by (*auto simp add*: *pair-less-def max-strict-def elim*!:*max-ext.cases intro*: *one-step-implies-mult*)

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

lemma *pw-leq-lstep*:
(*x*, *y*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
by (*drule pw-leq-step*) (*rule pw-leq-empty*, *simp*)

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-of A*, *set-of B*) ∈ *max-strict*
∨ (*B* = {#} ∧ *A* = {#}))
using *assms*
proof (*induct*)
case *pw-leq-empty* **thus** *?case* **by** *auto*
next
case (*pw-leq-step x y X Y*)
then obtain *A B Z* **where**
[*simp*]: *X* = *A* + *Z Y* = *B* + *Z*
and *1*[*simp*]: (*set-of A*, *set-of B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#})
by *auto*
from *pw-leq-step* **have** *x* = *y* ∨ (*x*, *y*) ∈ *pair-less*
unfolding *pair-leq-def* **by** *auto*
thus *?case*
proof
assume [*simp*]: *x* = *y*
have
{#*x*#} + *X* = *A* + ({#*y*#}+*Z*)
∧ {#*y*#} + *Y* = *B* + ({#*y*#}+*Z*)
∧ ((*set-of A*, *set-of B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#}))
by (*auto simp*: *add-ac*)
thus *?case* **by** (*intro exI*)
next
assume *A*: (*x*, *y*) ∈ *pair-less*

```
    let ?A' = {#x#} + A and ?B' = {#y#} + B
    have {#x#} + X = ?A' + Z
      {#y#} + Y = ?B' + Z
      by (auto simp add: add-ac)
    moreover have
      (set-of ?A', set-of ?B') ∈ max-strict
      using 1 A unfolding max-strict-def
      by (auto elim!: max-ext.cases)
    ultimately show ?thesis by blast
  qed
qed

lemma
  assumes pwleq: pw-leq Z Z'
  shows ms-strictI: (set-of A, set-of B) ∈ max-strict ⟹ (Z + A, Z' + B) ∈
ms-strict
  and   ms-weakI1: (set-of A, set-of B) ∈ max-strict ⟹ (Z + A, Z' + B) ∈
ms-weak
  and   ms-weakI2: (Z + {#}, Z' + {#}) ∈ ms-weak
proof −
  from pw-leq-split[OF pwleq]
  obtain A' B' Z''
    where [simp]: Z = A' + Z'' Z' = B' + Z''
   and mx-or-empty: (set-of A', set-of B') ∈ max-strict ∨ (A' = {#} ∧ B' = {#})
    by blast
  {
    assume max: (set-of A, set-of B) ∈ max-strict
    from mx-or-empty
    have (Z'' + (A + A'), Z'' + (B + B')) ∈ ms-strict
    proof
      assume max': (set-of A', set-of B') ∈ max-strict
      with max have (set-of (A + A'), set-of (B + B')) ∈ max-strict
        by (auto simp: max-strict-def intro: max-ext-additive)
      thus ?thesis by (rule smsI)
    next
      assume [simp]: A' = {#} ∧ B' = {#}
      show ?thesis by (rule smsI) (auto intro: max)
    qed
    thus (Z + A, Z' + B) ∈ ms-strict by (simp add:add-ac)
    thus (Z + A, Z' + B) ∈ ms-weak by (simp add: ms-weak-def)
  }
  from mx-or-empty
  have (Z'' + A', Z'' + B') ∈ ms-weak by (rule wmsI)
  thus (Z + {#}, Z' + {#}) ∈ ms-weak by (simp add:add-ac)
qed

lemma empty-idemp: {#} + x = x x + {#} = x
and nonempty-plus: {# x #} + rs ≠ {#}
and nonempty-single: {# x #} ≠ {#}
```

by *auto*

setup ⟪
let
fun msetT T = Type (Multiset.multiset, [T]);

fun mk-mset T [] = Const (@{const-name Mempty}, msetT T)
| mk-mset T [x] = Const (@{const-name single}, T --> msetT T) $ x
| mk-mset T (x :: xs) =
Const (@{const-name plus}, msetT T --> msetT T --> msetT T) $
mk-mset T [x] $ mk-mset T xs

fun mset-member-tac m i =
(if m <= 0 then
rtac @{thm multi-member-this} i ORELSE rtac @{thm multi-member-last}
i
else
rtac @{thm multi-member-skip} i THEN mset-member-tac (m − 1) i)

val mset-nonempty-tac =
rtac @{thm nonempty-plus} ORELSE' rtac @{thm nonempty-single}

val regroup-munion-conv =
FundefLib.regroup-conv @{const-name Multiset.Mempty} @{const-name plus}
(map (fn t => t RS eq-reflection) (@{thms union-ac} @ @{thms empty-idemp}))

fun unfold-pwleq-tac i =
(rtac @{thm pw-leq-step} i THEN (fn st => unfold-pwleq-tac (i + 1) st))
ORELSE (rtac @{thm pw-leq-lstep} i)
ORELSE (rtac @{thm pw-leq-empty} i)

val set-of-simps = [@{thm set-of-empty}, @{thm set-of-single}, @{thm set-of-union},
@{thm Un-insert-left}, @{thm Un-empty-left}]
in
ScnpReconstruct.multiset-setup (ScnpReconstruct.Multiset
{
msetT=msetT, mk-mset=mk-mset, mset-regroup-conv=regroup-munion-conv,
mset-member-tac=mset-member-tac, mset-nonempty-tac=mset-nonempty-tac,
mset-pwleq-tac=unfold-pwleq-tac, set-of-simps=set-of-simps,
smsI'= @{thm ms-strictI}, wmsI2''= @{thm ms-weakI2}, wmsI1= @{thm
ms-weakI1},
reduction-pair= @{thm ms-reduction-pair}
})
end
⟫

end

theory *LProd*
imports *Multiset*
begin

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

lemma (*as*,*bs*) ∈ *lprod R* ⟹ *length as* = *length bs*
  apply (*induct as bs rule*: *lprod.induct*)
  apply *auto*
  done

lemma (*as*, *bs*) ∈ *lprod R* ⟹ *1* ≤ *length as* ∧ *1* ≤ *length bs*
  apply (*induct as bs rule*: *lprod.induct*)
  apply *auto*
  done

lemma *lprod-subset-elem*: (*as*, *bs*) ∈ *lprod S* ⟹ *S* ⊆ *R* ⟹ (*as*, *bs*) ∈ *lprod R*
  apply (*induct as bs rule*: *lprod.induct*)
  apply (*auto*)
  done

lemma *lprod-subset*: *S* ⊆ *R* ⟹ *lprod S* ⊆ *lprod R*
  by (*auto intro*: *lprod-subset-elem*)

lemma *lprod-implies-mult*: (*as*, *bs*) ∈ *lprod R* ⟹ *trans R* ⟹ (*multiset-of as*, *multiset-of bs*) ∈ *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*: *algebra-simps*)
  have *bs*: *multiset-of* (*bh* @ *b* # *bt*) = *multiset-of* (*bh* @ *bt*) + {#*b*#} (**is** - = *?mb* + -)
    by (*simp add*: *algebra-simps*)
  from *prems* **have** (*?ma*, *?mb*) ∈ *mult R*
    by *auto*

```
  with mult-implies-one-step[OF transR] have
   ∃ I J K. ?mb = I + J ∧ ?ma = I + K ∧ J ≠ {#} ∧ (∀ k∈set-of K. ∃ j∈set-of
J. (k, j) ∈ R)
    by blast
  then obtain I J K where
     decomposed: ?mb = I + J ∧ ?ma = I + K ∧ J ≠ {#} ∧ (∀ k∈set-of K.
∃ j∈set-of J. (k, j) ∈ R)
    by blast
  show ?case
  proof (cases a = b)
    case True
    have ((I + {#b#}) + K, (I + {#b#}) + J) ∈ 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: algebra-simps)
      done
  next
    case False
    from False lprod-list have False: (a, b) ∈ R by blast
    have (I + (K + {#a#}), I + (J + {#b#})) ∈ 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: algebra-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^+) ⊆ inv-image (mult (R^+)) multiset-of
    by (auto simp add: lprod-implies-mult trans-trancl)
 note lprodtrancl = wf-subset[OF wf-inv-image[where r=mult (R^+) 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, sim-
plified]
  show ?thesis by (auto intro: lprod)
qed
```

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

lemma *lprod-2-3*: $(a, b) \in R \Longrightarrow ([a, c], [b, c]) \in lprod\ R$
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* ⇒ *ZF set*
  fixgames A ≡ { *Opair l r* | *l r*. *explode l* ⊆ *A* & *explode r* ⊆ *A*}
  games-lfp :: *ZF set*
  games-lfp ≡ *lfp fixgames*
  games-gfp :: *ZF set*
  games-gfp ≡ *gfp fixgames*

lemma *mono-fixgames*: *mono* (*fixgames*)
  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*^+)
proof −
{
fix *opt*
fix *g*

```
    assume (opt, g) ∈ is-option-of
   then have ∃ u v. (opt, u) ∈ (is-Elem-of^+) ∧ (u,v) ∈ (is-Elem-of^+) ∧ (v,g)
∈ (is-Elem-of^+)
     apply −
     apply (drule option2elem)
     apply (auto simp add: r-into-trancl' is-Elem-of-def)
     done
   then have (opt, g) ∈ (is-Elem-of^+)
     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^+) 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* ∈ *games-lfp*
  and *opt*: *left-option g opt* ∨ *right-option g opt*
  shows *opt* ∈ *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*) ∈ *is-option-of*
  shows *opt* ∈ *games-lfp* ∧ *g* ∈ *games-lfp*
proof −
  from *hyp* **have** *g-game*: *g* ∈ *games-lfp*
    by (*simp add*: *is-option-of-def games-lfp-eq-gfp*)
  from *hyp* **have** *left-option g opt* ∨ *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* ∈ *games-lfp* ⟹ ∃ *l r*. *x* = *Opair l r*
  apply (*rule exI*[**where** *x*=*Fst x*])
  apply (*rule exI*[**where** *x*=*Snd x*])
  apply (*subgoal-tac x* ∈ (*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* ⇒ *game zet*
left-options g ≡ *zimage Abs-game* (*zexplode* (*Fst* (*Rep-game g*)))
right-options :: *game* ⇒ *game zet*
right-options g ≡ *zimage Abs-game* (*zexplode* (*Snd* (*Rep-game g*)))
options :: *game* ⇒ *game zet*
options g ≡ *zunion* (*left-options g*) (*right-options g*)
Game :: *game zet* ⇒ *game zet* ⇒ *game*
Game L R ≡ *Abs-game* (*Opair* (*zimplode* (*zimage Rep-game L*)) (*zimplode* (*zimage Rep-game R*)))

lemma *Repl-Rep-game-Abs-game*: ∀ *e*. *Elem e z* ⟶ *e* ∈ *games-lfp* ⟹ *Repl z* (*Rep-game o Abs-game*) = *z*
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-eq: 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-eq, 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-eq, auto)
    qed
  }
  note left = this
  from left right show ?case
    by (auto, subst ge-game-eq, 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-eq, 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-eq, auto)
      with goal2 show ?case by auto}
    note left = this
    {case goal3
      from left right show ?case
        by (subst ge-game-eq, 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: (⋀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-eq[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-eq[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-eq[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-eq[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-eq[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-eq)
       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-eq*[*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-eq*[*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-eq)
     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 ∀ x y. a = [x, y] ⟶ 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-eq*[*of neg-game x neg-game y*] *ge-game-eq*[*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-on-def sym-def trans-def eq-game-rel-def*
eq-game-sym intro: *eq-game-refl eq-game-trans*)

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

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

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

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

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

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

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

instance ..

end

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}) ≤ 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]: (∀ g. z = Abs-Pg (eq-game-rel ``
{g}) ⟶ P) ⟶ 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 + (- b) by (simp add: Pg-diff-def)
  {
    assume ab: a ≤ b
    assume ba: b ≤ a
    from ab ba show a = b
      apply (cases a, cases b)
      apply (simp add: eq-game-def)
      done
  }
  then show (a < b) = (a ≤ b ∧ ¬ b ≤ a) by (auto simp add: Pg-less-def)
  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
```