Separation in Generic Extensions

Emmanuel Gunther   Miguel Pagano   Pedro Sánchez Terraf

January 10, 2019

Contents

theory *Pointed-DC* **imports** *ZF.AC*

begin

This proof of DC is from Moschovakis "Notes on Set Theory"

```
consts dc-witness :: i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i
primrec
  wit0   : dc-witness(0,A,a,s,R) = a
  witrec :dc-witness(succ(n),A,a,s,R) = s`{x∈A. ⟨dc-witness(n,A,a,s,R),x⟩∈R }

lemma witness-into-A [TC]:  a∈A ⟹ n∈nat ⟹
                             (∀ X . X≠0 ∧ X⊆A ⟶ s`X∈X) ⟹
                             ∀ y∈A. {x∈A. ⟨y,x⟩∈R } ≠ 0 ⟹
                             dc-witness(n, A, a, s, R)∈A
  apply (induct-tac n ,simp+)
  apply (drule-tac x=dc-witness(x, A, a, s, R) in bspec, assumption)
  apply (drule-tac x={xa ∈ A . ⟨dc-witness(x, A, a, s, R), xa⟩ ∈ R} in  spec)
  apply auto
  done
lemma witness-related :  a∈A ⟹ n∈nat ⟹
                             (∀ X . X≠0 ∧ X⊆A ⟶ s`X∈X) ⟹
                             ∀ y∈A. {x∈A. ⟨y,x⟩∈R } ≠ 0 ⟹
                                ⟨dc-witness(n, A, a, s, R),dc-witness(succ(n), A, a, s,
R)⟩∈R
  apply (frule-tac n=n and s=s  and R=R in witness-into-A, assumption+)
   apply (drule-tac x=dc-witness(n, A, a, s, R) in bspec, assumption)
  apply (drule-tac x= {x ∈ A . ⟨dc-witness(n, A, a, s, R), x⟩ ∈ R} in spec)
  apply (simp, blast)
  done

lemma witness-funtype: a∈A ⟹
                        (∀ X . X≠0 ∧ X⊆A ⟶ s`X∈X) ⟹
                        ∀ y∈A. {x∈A. ⟨y,x⟩∈R } ≠ 0 ⟹
                        (λn∈nat. dc-witness(n, A, a, s, R)) ∈ nat → A
  apply (rule-tac B={dc-witness(n, A, a, s, R). n∈nat} in fun-weaken-type)
```

```
  apply (rule lam-funtype)
  apply ( blast intro:witness-into-A)
  done

lemma witness-to-fun: a∈A ⟹ (∀ X . X≠0 ∧ X⊆A ⟶ s‘X∈X) ⟹
                      ∀ y∈A. {x∈A. ⟨y,x⟩∈R } ≠ 0 ⟹
                      ∃ f ∈ nat→A. ∀ n∈nat. f‘n =dc-witness(n,A,a,s,R)
  apply (rule-tac x=λn∈nat. dc-witness(n,A,a,s,R) in  bexI, simp)
  apply (rule witness-funtype, simp+)
  done

theorem pointed-DC  : (∀ x∈A. ∃ y∈A. ⟨x,y⟩∈ R) ⟹
                  ∀ a∈A. (∃ f ∈ nat→A. f‘0 = a ∧ (∀ n ∈ nat. ⟨f‘n,f‘succ(n)⟩∈R))
  apply (rule)
  apply (insert AC-func-Pow)
  apply (drule allI)
  apply (drule-tac x=A in spec)
  apply (drule-tac P=λf .∀ x∈Pow(A) − {0}. f ‘ x ∈ x
             and A=Pow(A)−{0}→ A
             and Q= ∃ f∈nat→ A. f ‘ 0 = a ∧ (∀ n∈nat. ⟨f ‘ n, f ‘ succ(n)⟩ ∈ R)
in bexE)
  prefer 2 apply (assumption)
  apply (rename-tac s)
  apply (rule-tac x=λn∈nat. dc-witness(n,A,a,s,R) in bexI)
   prefer 2  apply (blast intro:witness-funtype)
  apply (rule  conjI, simp)
  apply (rule ballI, rename-tac m)
  apply (subst beta, simp)+
  apply (rule witness-related, auto)
  done


lemma aux-DC-on-AxNat2 : ∀ x∈A×nat. ∃ y∈A. ⟨x,⟨y,succ(snd(x))⟩⟩ ∈ R ⟹
              ∀ x∈A×nat. ∃ y∈A×nat. ⟨x,y⟩ ∈ {⟨a,b⟩∈R. snd(b) = succ(snd(a))}
  apply (rule ballI, erule-tac x=x in ballE, simp-all)
  done

lemma infer-snd : c∈ A×B ⟹ snd(c) = k ⟹ c=⟨fst(c),k⟩
  by auto

corollary DC-on-A-x-nat :
  (∀ x∈A×nat. ∃ y∈A. ⟨x,⟨y,succ(snd(x))⟩⟩ ∈ R) ⟹
    ∀ a∈A. (∃ f ∈ nat→A. f‘0 = a ∧ (∀ n ∈ nat. ⟨⟨f‘n,n⟩,⟨f‘succ(n),succ(n)⟩⟩∈R))
  apply (frule aux-DC-on-AxNat2)
  apply (drule-tac R={⟨a,b⟩∈R. snd(b) = succ(snd(a))} in  pointed-DC)
  apply (rule ballI)
  apply (rotate-tac)
  apply (drule-tac x=⟨a,0⟩ in  bspec, simp)
  apply (erule bexE, rename-tac g)
```

apply (*rule-tac* $x{=}\lambda x{\in}nat.\ fst(g`x)$ **and** $A{=}nat{\rightarrow}A$ **in** *bexI*, *auto*)
apply (*subgoal-tac* $\forall\, n{\in}nat.\ g`n{=}\ \langle fst(g\ `\ n),\ n\rangle$)
prefer *2* **apply** (*rule ballI*, *rename-tac m*)
apply (*induct-tac m*, *simp*)
apply (*rename-tac d*, *auto*)
apply (*frule-tac* $A{=}nat$ **and** $x{=}d$ **in** *bspec*, *simp*)
apply (*rule-tac* $A{=}A$ **and** $B{=}nat$ **in** *infer-snd*, *auto*)
apply (*rule-tac* $a{=}\langle fst(g\ `\ d),\ d\rangle$ **and** $b{=}g\ `\ d$ **in** *ssubst*, *assumption*)

apply (*subst snd-conv*, *simp*)
done

lemma *aux-sequence-DC* : $\bigwedge R.\ \forall\, x{\in}A.\ \forall\, n{\in}nat.\ \exists\, y{\in}A.\ \langle x,y\rangle \in S`n \Longrightarrow$
$R{=}\{\langle\langle x,n\rangle,\langle y,m\rangle\rangle{\in}(A{\times}nat){\times}(A{\times}nat).\ \langle x,y\rangle{\in}S`m\ \} \Longrightarrow$
$\forall\, x{\in}A{\times}nat.\ \exists\, y{\in}A.\ \langle x,\langle y,succ(snd(x))\rangle\rangle \in R$
apply (*rule ballI*, *rename-tac v*)
apply (*frule Pair-fst-snd-eq*)
apply (*erule-tac* $x{=}fst(v)$ **in** *ballE*)
apply (*drule-tac* $x{=}succ(snd(v))$ **in** *bspec*, *auto*)
done

lemma *aux-sequence-DC2* : $\forall\, x{\in}A.\ \forall\, n{\in}nat.\ \exists\, y{\in}A.\ \langle x,y\rangle \in S`n \Longrightarrow$
$\forall\, x{\in}A{\times}nat.\ \exists\, y{\in}A.\ \langle x,\langle y,succ(snd(x))\rangle\rangle \in \{\langle\langle x,n\rangle,\langle y,m\rangle\rangle{\in}(A{\times}nat){\times}(A{\times}nat).$
$\langle x,y\rangle{\in}S`m\ \}$
by *auto*

lemma *sequence-DC*: $\forall\, x{\in}A.\ \forall\, n{\in}nat.\ \exists\, y{\in}A.\ \langle x,y\rangle \in S`n \Longrightarrow$
$\forall\, a{\in}A.\ (\exists f \in nat{\rightarrow}A.\ f`0 = a \wedge (\forall\, n \in nat.\ \langle f`n,f`succ(n)\rangle{\in}S`succ(n)))$
apply (*drule aux-sequence-DC2*)
apply (*drule DC-on-A-x-nat*, *auto*)
done
end
theory *Forcing-Notions* **imports** *Pointed-DC* **begin**

definition *compat-in* :: $i{\Rightarrow}i{\Rightarrow}i{\Rightarrow}i{\Rightarrow}o$ **where**
$compat\text{-}in(A,r,p,q) == \exists\, d{\in}A\ .\ \langle d,p\rangle{\in}r \wedge \langle d,q\rangle{\in}r$

lemma *compat-inI* :
$[\![\ d{\in}A\ ;\ \langle d,p\rangle{\in}r\ ;\ \langle d,g\rangle{\in}r\ ]\!] \Longrightarrow compat\text{-}in(A,r,p,g)$
by (*auto simp add*: *compat-in-def*)

lemma *refl-compat*:
$[\![\ refl(A,r)\ ;\ \langle p,q\rangle \in r\ \mid\ p{=}q\ \mid\ \langle q,p\rangle \in r\ ;\ p{\in}A\ ;\ q{\in}A]\!] \Longrightarrow compat\text{-}in(A,r,p,q)$
by (*auto simp add*: *refl-def compat-inI*)

lemma *chain-compat*:
$refl(A,r) \Longrightarrow linear(A,r) \Longrightarrow\ (\forall\, p{\in}A.\forall\, q{\in}A.\ compat\text{-}in(A,r,p,q))$
by (*simp add*: *refl-compat linear-def*)

lemma *subset-fun-image*: $f:N \to P \Longrightarrow f``N \subseteq P$
by (*auto simp add*: *image-fun apply-funtype*)

definition
antichain :: $i \Rightarrow i \Rightarrow i \Rightarrow o$ **where**
$antichain(P,leq,A) == A \subseteq P \wedge (\forall p \in A. \forall q \in A. (\neg\ compat\text{-}in(P,leq,p,q)))$

definition
ccc :: $i \Rightarrow i \Rightarrow o$ **where**
$ccc(P,leq) == \forall A.\ antichain(P,leq,A) \longrightarrow |A| \leq nat$

locale *forcing-notion* =
fixes *P leq one*
assumes *one-in-P*: $one \in P$
and *leq-preord*: $preorder\text{-}on(P,leq)$
and *one-max*: $\forall p \in P.\ \langle p,one \rangle \in leq$
begin
definition
dense :: $i \Rightarrow o$ **where**
$dense(D) == \forall p \in P.\ \exists d \in D\ .\ \langle d,p \rangle \in leq$

definition
dense-below :: $i \Rightarrow i \Rightarrow o$ **where**
$dense\text{-}below(D,q) == \forall p \in P.\ \langle p,q \rangle \in leq \longrightarrow (\exists d \in D\ .\ \langle d,p \rangle \in leq)$

lemma *P-dense*: *dense*(*P*)
by (*insert leq-preord*, *auto simp add*: *preorder-on-def refl-def dense-def*)

definition
increasing :: $i \Rightarrow o$ **where**
$increasing(F) == \forall x \in F.\ \forall\ p \in P\ .\ \langle x,p \rangle \in leq \longrightarrow p \in F$

definition
compat :: $i \Rightarrow i \Rightarrow o$ **where**
$compat(p,q) == compat\text{-}in(P,leq,p,q)$

definition
antichain :: $i \Rightarrow o$ **where**
$antichain(A) == A \subseteq P \wedge (\forall p \in A. \forall q \in A. (\neg compat(p,q)))$

definition
filter :: $i \Rightarrow o$ **where**
$filter(G) == G \subseteq P \wedge increasing(G) \wedge (\forall p \in G.\ \forall q \in G.\ compat\text{-}in(G,leq,p,q))$

lemma *filterD* : $filter(G) \Longrightarrow x \in G \Longrightarrow x \in P$
by (*auto simp add* : *subsetD filter-def*)

lemma *filter-leqD* : $filter(G) \Longrightarrow x \in G \Longrightarrow y \in P \Longrightarrow <x,y> \in leq \Longrightarrow y \in G$
by (*simp add*: *filter-def increasing-def*)

lemma *low-bound-filter* :
assumes *filter*(*G*) **and** $p \in G$ **and** $q \in G$
shows $\exists\, r \in G.\ <r,p> \in leq \wedge <r,q> \in leq$
using *assms*
unfolding *compat-in-def filter-def* **by** *blast*

definition
upclosure :: $i \Rightarrow i$ **where**
$upclosure(A) == \{p \in P. \exists\, a \in A. \langle a,p\rangle \in leq\}$

lemma *upclosureI* [*intro*] : $p \in P \Longrightarrow a \in A \Longrightarrow \langle a,p\rangle \in leq \Longrightarrow p \in upclosure(A)$
by (*simp add*:*upclosure-def*, *auto*)

lemma *upclosureE* [*elim*] :
$p \in upclosure(A) \Longrightarrow (\bigwedge x\ a.\ x \in P \Longrightarrow a \in A \Longrightarrow \langle a,x\rangle \in leq \Longrightarrow R) \Longrightarrow R$
by (*auto simp add*:*upclosure-def*)

lemma *upclosureD* [*dest*] :
$p \in upclosure(A) \Longrightarrow \exists\, a \in A.(\langle a,p\rangle \in leq) \wedge p \in P$
by (*simp add*:*upclosure-def*)

lemma *upclosure-increasing* :
$A \subseteq P \Longrightarrow increasing(upclosure(A))$
apply (*unfold increasing-def upclosure-def*, *simp*)
apply *clarify*
apply (*rule-tac x=a* **in** *bexI*)
apply (*insert leq-preord*, *unfold preorder-on-def*)
apply (*drule conjunct2*, *unfold trans-on-def*)
apply (*drule-tac x=a* **in** *bspec*, *fast*)
apply (*drule-tac x=x* **in** *bspec*, *assumption*)
apply (*drule-tac x=p* **in** *bspec*, *assumption*)
apply (*simp*, *assumption*)
done

lemma *upclosure-in-P*: $A \subseteq P \Longrightarrow upclosure(A) \subseteq P$
apply (*rule subsetI*)
apply (*simp add*:*upclosure-def*)
done

lemma *A-sub-upclosure*: $A \subseteq P \Longrightarrow A \subseteq upclosure(A)$
apply (*rule subsetI*)
apply (*simp add*:*upclosure-def*, *auto*)
apply (*insert leq-preord*, *unfold preorder-on-def refl-def*, *auto*)
done

lemma *elem-upclosure*: $A \subseteq P \Longrightarrow x \in A \Longrightarrow x \in upclosure(A)$
by (*blast dest*:*A-sub-upclosure*)

```
lemma closure-compat-filter:
  A⊆P ⟹ (∀p∈A.∀q∈A. compat-in(A,leq,p,q)) ⟹ filter(upclosure(A))
  apply (unfold filter-def)
  apply (intro conjI)
  apply (rule upclosure-in-P, assumption)
   apply (rule upclosure-increasing, assumption)
  apply (unfold compat-in-def)
  apply (rule ballI)+
  apply (rename-tac x y)
  apply (drule upclosureD)+
  apply (erule bexE)+
  apply (rename-tac a b)
  apply (drule-tac A=A
              and x=a in bspec, assumption)
  apply (drule-tac A=A
              and x=b in bspec, assumption)
  apply (auto)
  apply (rule-tac x=d in bexI)
  prefer 2 apply (simp add:A-sub-upclosure [THEN subsetD])
  apply (insert leq-preord, unfold preorder-on-def trans-on-def,  drule conjunct2)
  apply (rule conjI)
  apply (drule-tac x=d in bspec, rule-tac A=A in subsetD, assumption+)
  apply (drule-tac x=a in bspec, rule-tac A=A in subsetD, assumption+)
  apply (drule-tac x=x in bspec, assumption, auto)
  done

lemma aux-RS1: f ∈ N → P ⟹ n∈N ⟹ f`n ∈ upclosure(f ``N)
  apply (rule-tac  elem-upclosure)
   apply (rule subset-fun-image, assumption)
  apply (simp add: image-fun, blast)
  done
end

lemma refl-monot-domain: refl(B,r) ⟹ A⊆B ⟹ refl(A,r)
  apply (drule subset-iff [THEN iffD1])
  apply (unfold refl-def)
  apply (blast)
  done

lemma decr-succ-decr: f ∈ nat → P ⟹ preorder-on(P,leq) ⟹
        ∀n∈nat. ⟨f ` succ(n), f ` n⟩ ∈ leq ⟹
          n∈nat ⟹ m∈nat ⟹ n≤m ⟶ ⟨f ` m, f ` n⟩ ∈ leq
  apply (unfold preorder-on-def, erule conjE)
  apply (induct-tac m, simp add:refl-def, rename-tac x)
  apply (rule impI)
  apply (case-tac n≤x, simp)
    apply (drule-tac x=x in bspec, assumption)
   apply (unfold trans-on-def)
   apply (drule-tac x=f`succ(x) in bspec, simp)
```

apply (*drule-tac x=f'x* **in** *bspec*, *simp*)
apply (*drule-tac x=f'n* **in** *bspec*, *auto*)
apply (*drule-tac le-succ-iff* [*THEN iffD1*], *simp add*: *refl-def*)
done

lemma *not-le-imp-lt*: $[\![\sim i \leq j ; Ord(i); Ord(j)]\!] \Longrightarrow j<i$
by (*simp add*:*not-le-iff-lt*)

lemma *decr-seq-linear*: $refl(P,leq) \Longrightarrow f \in nat \to P \Longrightarrow$
$\forall n\in nat. \langle f ` succ(n), f ` n\rangle \in leq \Longrightarrow$
$trans[P](leq) \Longrightarrow linear(f `` nat, leq)$
apply (*unfold linear-def*)
apply (*rule ball-image-simp* [*THEN iffD2*], *assumption*, *simp*, *rule ballI*)+
apply (*rename-tac y*)
apply (*case-tac x≤y*)
apply (*drule-tac leq=leq* **and** *n=x* **and** *m=y* **in** *decr-succ-decr*)

apply (*simp add*:*preorder-on-def*)

apply (*simp+*)
apply (*drule not-le-imp-lt* [*THEN leI*], *simp-all*)
apply (*drule-tac leq=leq* **and** *n=y* **and** *m=x* **in** *decr-succ-decr*)

apply (*simp add*:*preorder-on-def*)

apply (*simp+*)
done

locale *countable-generic* = *forcing-notion* +
fixes $\mathcal{D}$
assumes *countable-subs-of-P*: $\mathcal{D} \in nat\to Pow(P)$
and *seq-of-denses*: $\forall n \in nat. dense(\mathcal{D}`n)$

begin

definition
D-generic :: $i\Rightarrow o$ **where**
$D\text{-}generic(G) == filter(G) \wedge (\forall n\in nat.(\mathcal{D}`n)\cap G\neq 0)$

lemma *RS-relation*:
assumes
1: $x\in P$
and
2: $n\in nat$
shows
$\exists y\in P. \langle x,y\rangle \in (\lambda m\in nat. \{\langle x,y\rangle\in P*P. \langle y,x\rangle\in leq \wedge y\in\mathcal{D}`(pred(m))\})`n$
proof −

from *seq-of-denses* **and** *2* **have** *dense*($\mathcal{D}$ ‘ *pred*(*n*)) **by** (*simp*)
with *1* **have**
$\exists\, d{\in}\mathcal{D}$ ‘ *Arith.pred*(*n*). ⟨*d*, *x*⟩ ∈ *leq*
unfolding *dense-def* **by** (*simp*)
then obtain *d* **where**
3: *d* ∈ $\mathcal{D}$ ‘ *Arith.pred*(*n*) ∧ ⟨*d*, *x*⟩ ∈ *leq*
by (*rule bexE*, *simp*)
from *countable-subs-of-P* **have**
$\mathcal{D}$ ‘ *Arith.pred*(*n*) ∈ *Pow*(*P*)
using *2* **by** (*blast dest*:*apply-funtype intro*:*pred-type*)
then have
$\mathcal{D}$ ‘ *Arith.pred*(*n*) ⊆ *P*
by (*rule PowD*)
then have
d ∈ *P* ∧ ⟨*d*, *x*⟩ ∈ *leq* ∧ *d* ∈ $\mathcal{D}$ ‘ *Arith.pred*(*n*)
using *3* **by** *auto*
then show *?thesis* **using** *1* **and** *2* **by** *auto*
qed

theorem *rasiowa-sikorski*:
p∈*P* ⟹ ∃ *G*. *p*∈*G* ∧ *D-generic*(*G*)
proof −
assume
Eq1: *p*∈*P*
let
?S=(λ*m*∈*nat*. {⟨*x*,*y*⟩∈*P*∗*P*. ⟨*y*,*x*⟩∈*leq* ∧ *y*∈$\mathcal{D}$‘(*pred*(*m*))})
from *RS-relation* **have**
∀ *x*∈*P*. ∀ *n*∈*nat*. ∃ *y*∈*P*. ⟨*x*,*y*⟩ ∈ *?S*‘*n*
by (*auto*)
with *sequence-DC* **have**
∀ *a*∈*P*. (∃ *f* ∈ *nat*→*P*. *f*‘*0* = *a* ∧ (∀ *n* ∈ *nat*. ⟨*f*‘*n*,*f*‘*succ*(*n*)⟩∈*?S*‘*succ*(*n*)))
by (*blast*)
then obtain *f* **where**
Eq2: *f* : *nat*→*P*
and
Eq3: *f* ‘ *0* = *p* ∧
(∀ *n*∈*nat*.
f ‘ *n* ∈ *P* ∧ *f* ‘ *succ*(*n*) ∈ *P* ∧ ⟨*f* ‘ *succ*(*n*), *f* ‘ *n*⟩ ∈ *leq* ∧
f ‘ *succ*(*n*) ∈ $\mathcal{D}$ ‘ *n*)
using *Eq1* **by** (*auto*)
then have
Eq4: *f*‘‘*nat* ⊆ *P*
by (*simp add*:*subset-fun-image*)
with *leq-preord* **have**
Eq5: *refl*(*f*‘‘*nat*, *leq*) ∧ *trans*[*P*](*leq*)
unfolding *preorder-on-def* **by** (*blast intro*:*refl-monot-domain*)
from *Eq3* **have**
∀ *n*∈*nat*. ⟨*f* ‘ *succ*(*n*), *f* ‘ *n*⟩ ∈ *leq*
by (*simp*)

with *Eq2* **and** *Eq5* **and** *leq-preord* **and** *decr-seq-linear* **have**
Eq6: *linear*(*f``nat*, *leq*)
unfolding *preorder-on-def* **by** (*blast*)
with *Eq5* **and** *chain-compat* **have**
(∀ *p*∈*f``nat*.∀ *q*∈*f``nat*. *compat-in*(*f``nat*,*leq*,*p*,*q*))
by (*auto*)
then have
fil: *filter*(*upclosure*(*f``nat*))
(**is** *filter*(*?G*))
using *closure-compat-filter* **and** *Eq4* **by** *simp*
have
gen: ∀ *n*∈*nat*. $\mathcal{D}$ ` *n* ∩ *?G* ≠ *0*
proof
fix *n*
assume
n∈*nat*
with *Eq2* **and** *Eq3* **have**
f`succ*(*n*) ∈ *?G* ∧ *f`succ(*n*) ∈ $\mathcal{D}$ ` *n*
using *aux-RS1* **by** *simp*
then show
$\mathcal{D}$ ` *n* ∩ *?G* ≠ *0*
by *blast*
qed
from *Eq3* **and** *Eq2* **have**
p ∈ *?G*
using *aux-RS1* **by** *auto*
with *gen* **and** *fil* **show** *?thesis*
unfolding *D-generic-def* **by** *auto*
qed

end

end
theory *Relative* **imports** *ZF* **begin**

1 Relativization and Absoluteness

1.1 Relativized versions of standard set-theoretic concepts

definition
empty :: [*i*=>*o*,*i*] => *o* **where**
empty(*M*,*z*) == ∀ *x*[*M*]. *x* ∉ *z*

definition
subset :: [*i*=>*o*,*i*,*i*] => *o* **where**
subset(*M*,*A*,*B*) == ∀ *x*[*M*]. *x*∈*A* ⟶ *x* ∈ *B*

definition
upair :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**

$upair(M,a,b,z) == a \in z \ \& \ b \in z \ \& \ (\forall x[M]. \ x{\in}z \longrightarrow x = a \mid x = b)$

definition
$pair$ :: $[i{=}{>}o,i,i,i] => o$ **where**
$pair(M,a,b,z) == \exists x[M]. \ upair(M,a,a,x) \ \&$
$(\exists y[M]. \ upair(M,a,b,y) \ \& \ upair(M,x,y,z))$

definition
$union$ :: $[i{=}{>}o,i,i,i] => o$ **where**
$union(M,a,b,z) == \forall x[M]. \ x \in z \longleftrightarrow x \in a \mid x \in b$

definition
$is\text{-}cons$ :: $[i{=}{>}o,i,i,i] => o$ **where**
$is\text{-}cons(M,a,b,z) == \exists x[M]. \ upair(M,a,a,x) \ \& \ union(M,x,b,z)$

definition
$successor$ :: $[i{=}{>}o,i,i] => o$ **where**
$successor(M,a,z) == is\text{-}cons(M,a,a,z)$

definition
$number1$ :: $[i{=}{>}o,i] => o$ **where**
$number1(M,a) == \exists x[M]. \ empty(M,x) \ \& \ successor(M,x,a)$

definition
$number2$ :: $[i{=}{>}o,i] => o$ **where**
$number2(M,a) == \exists x[M]. \ number1(M,x) \ \& \ successor(M,x,a)$

definition
$number3$ :: $[i{=}{>}o,i] => o$ **where**
$number3(M,a) == \exists x[M]. \ number2(M,x) \ \& \ successor(M,x,a)$

definition
$powerset$ :: $[i{=}{>}o,i,i] => o$ **where**
$powerset(M,A,z) == \forall x[M]. \ x \in z \longleftrightarrow subset(M,x,A)$

definition
$is\text{-}Collect$ :: $[i{=}{>}o,i,i{=}{>}o,i] => o$ **where**
$is\text{-}Collect(M,A,P,z) == \forall x[M]. \ x \in z \longleftrightarrow x \in A \ \& \ P(x)$

definition
$is\text{-}Replace$ :: $[i{=}{>}o,i,[i,i]{=}{>}o,i] => o$ **where**
$is\text{-}Replace(M,A,P,z) == \forall u[M]. \ u \in z \longleftrightarrow (\exists x[M]. \ x{\in}A \ \& \ P(x,u))$

definition
$inter$ :: $[i{=}{>}o,i,i,i] => o$ **where**
$inter(M,a,b,z) == \forall x[M]. \ x \in z \longleftrightarrow x \in a \ \& \ x \in b$

definition

setdiff :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
setdiff(*M*,*a*,*b*,*z*) == $\forall$ *x*[*M*]. $x \in z \longleftrightarrow x \in a$ & $x \notin b$

definition
big-union :: [*i*=>*o*,*i*,*i*] => *o* **where**
big-union(*M*,*A*,*z*) == $\forall$ *x*[*M*]. $x \in z \longleftrightarrow$ ($\exists$ *y*[*M*]. $y \in A$ & $x \in y$)

definition
big-inter :: [*i*=>*o*,*i*,*i*] => *o* **where**
big-inter(*M*,*A*,*z*) ==
(*A*=*0* $\longrightarrow$ *z*=*0*) &
(*A*$\neq$*0* $\longrightarrow$ ($\forall$ *x*[*M*]. $x \in z \longleftrightarrow$ ($\forall$ *y*[*M*]. $y \in A \longrightarrow x \in y$)))

definition
cartprod :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
cartprod(*M*,*A*,*B*,*z*) ==
$\forall$ *u*[*M*]. $u \in z \longleftrightarrow$ ($\exists$ *x*[*M*]. $x \in A$ & ($\exists$ *y*[*M*]. $y \in B$ & *pair*(*M*,*x*,*y*,*u*)))

definition
is-sum :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
is-sum(*M*,*A*,*B*,*Z*) ==
$\exists$ *A0*[*M*]. $\exists$ *n1*[*M*]. $\exists$ *s1*[*M*]. $\exists$ *B1*[*M*].
number1(*M*,*n1*) & *cartprod*(*M*,*n1*,*A*,*A0*) & *upair*(*M*,*n1*,*n1*,*s1*) &
cartprod(*M*,*s1*,*B*,*B1*) & *union*(*M*,*A0*,*B1*,*Z*)

definition
is-Inl :: [*i*=>*o*,*i*,*i*] => *o* **where**
is-Inl(*M*,*a*,*z*) == $\exists$ *zero*[*M*]. *empty*(*M*,*zero*) & *pair*(*M*,*zero*,*a*,*z*)

definition
is-Inr :: [*i*=>*o*,*i*,*i*] => *o* **where**
is-Inr(*M*,*a*,*z*) == $\exists$ *n1*[*M*]. *number1*(*M*,*n1*) & *pair*(*M*,*n1*,*a*,*z*)

definition
is-converse :: [*i*=>*o*,*i*,*i*] => *o* **where**
is-converse(*M*,*r*,*z*) ==
$\forall$ *x*[*M*]. $x \in z \longleftrightarrow$
($\exists$ *w*[*M*]. $w \in r$ & ($\exists$ *u*[*M*]. $\exists$ *v*[*M*]. *pair*(*M*,*u*,*v*,*w*) & *pair*(*M*,*v*,*u*,*x*)))

definition
pre-image :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
pre-image(*M*,*r*,*A*,*z*) ==
$\forall$ *x*[*M*]. $x \in z \longleftrightarrow$ ($\exists$ *w*[*M*]. $w \in r$ & ($\exists$ *y*[*M*]. $y \in A$ & *pair*(*M*,*x*,*y*,*w*)))

definition
is-domain :: [*i*=>*o*,*i*,*i*] => *o* **where**
is-domain(*M*,*r*,*z*) ==
$\forall$ *x*[*M*]. $x \in z \longleftrightarrow$ ($\exists$ *w*[*M*]. $w \in r$ & ($\exists$ *y*[*M*]. *pair*(*M*,*x*,*y*,*w*)))

definition
$image$ :: $[i=>o,i,i,i] => o$ **where**
$image(M,r,A,z) ==$
$\forall y[M].\ y \in z \longleftrightarrow (\exists w[M].\ w\in r\ \&\ (\exists x[M].\ x\in A\ \&\ pair(M,x,y,w)))$

definition
$is\text{-}range$ :: $[i=>o,i,i] => o$ **where**
— the cleaner $\exists r'[M].\ is\text{-}converse(M, r, r') \wedge is\text{-}domain(M, r', z)$ unfortunately needs an instance of separation in order to prove $M(converse(r))$.
$is\text{-}range(M,r,z) ==$
$\forall y[M].\ y \in z \longleftrightarrow (\exists w[M].\ w\in r\ \&\ (\exists x[M].\ pair(M,x,y,w)))$

definition
$is\text{-}field$ :: $[i=>o,i,i] => o$ **where**
$is\text{-}field(M,r,z) ==$
$\exists dr[M].\ \exists rr[M].\ is\text{-}domain(M,r,dr)\ \&\ is\text{-}range(M,r,rr)\ \&$
$union(M,dr,rr,z)$

definition
$is\text{-}relation$ :: $[i=>o,i] => o$ **where**
$is\text{-}relation(M,r) ==$
$(\forall z[M].\ z\in r \longrightarrow (\exists x[M].\ \exists y[M].\ pair(M,x,y,z)))$

definition
$is\text{-}function$ :: $[i=>o,i] => o$ **where**
$is\text{-}function(M,r) ==$
$\forall x[M].\ \forall y[M].\ \forall y'[M].\ \forall p[M].\ \forall p'[M].$
$pair(M,x,y,p) \longrightarrow pair(M,x,y',p') \longrightarrow p\in r \longrightarrow p'\in r \longrightarrow y=y'$

definition
$fun\text{-}apply$ :: $[i=>o,i,i,i] => o$ **where**
$fun\text{-}apply(M,f,x,y) ==$
$(\exists xs[M].\ \exists fxs[M].$
$upair(M,x,x,xs)\ \&\ image(M,f,xs,fxs)\ \&\ big\text{-}union(M,fxs,y))$

definition
$typed\text{-}function$ :: $[i=>o,i,i,i] => o$ **where**
$typed\text{-}function(M,A,B,r) ==$
$is\text{-}function(M,r)\ \&\ is\text{-}relation(M,r)\ \&\ is\text{-}domain(M,r,A)\ \&$
$(\forall u[M].\ u\in r \longrightarrow (\forall x[M].\ \forall y[M].\ pair(M,x,y,u) \longrightarrow y\in B))$

definition
$is\text{-}funspace$ :: $[i=>o,i,i,i] => o$ **where**
$is\text{-}funspace(M,A,B,F) ==$
$\forall f[M].\ f \in F \longleftrightarrow typed\text{-}function(M,A,B,f)$

definition
$composition$ :: $[i=>o,i,i,i] => o$ **where**
$composition(M,r,s,t) ==$

$\forall\, p[M].\ p \in t \longleftrightarrow$
$(\exists\, x[M].\ \exists\, y[M].\ \exists\, z[M].\ \exists\, xy[M].\ \exists\, yz[M].$
$pair(M,x,z,p)\ \&\ pair(M,x,y,xy)\ \&\ pair(M,y,z,yz)\ \&$
$xy \in s\ \&\ yz \in r)$

definition
injection :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
$injection(M,A,B,f) ==$
$typed\text{-}function(M,A,B,f)\ \&$
$(\forall\, x[M].\ \forall\, x'[M].\ \forall\, y[M].\ \forall\, p[M].\ \forall\, p'[M].$
$pair(M,x,y,p) \longrightarrow pair(M,x',y,p') \longrightarrow p{\in}f \longrightarrow p'{\in}f \longrightarrow x{=}x')$

definition
surjection :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
$surjection(M,A,B,f) ==$
$typed\text{-}function(M,A,B,f)\ \&$
$(\forall\, y[M].\ y{\in}B \longrightarrow (\exists\, x[M].\ x{\in}A\ \&\ fun\text{-}apply(M,f,x,y)))$

definition
bijection :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
$bijection(M,A,B,f) == injection(M,A,B,f)\ \&\ surjection(M,A,B,f)$

definition
restriction :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
$restriction(M,r,A,z) ==$
$\forall\, x[M].\ x \in z \longleftrightarrow (x \in r\ \&\ (\exists\, u[M].\ u{\in}A\ \&\ (\exists\, v[M].\ pair(M,u,v,x))))$

definition
transitive-set :: [*i*=>*o*,*i*] => *o* **where**
$transitive\text{-}set(M,a) == \forall\, x[M].\ x{\in}a \longrightarrow subset(M,x,a)$

definition
ordinal :: [*i*=>*o*,*i*] => *o* **where**
— an ordinal is a transitive set of transitive sets
$ordinal(M,a) == transitive\text{-}set(M,a)\ \&\ (\forall\, x[M].\ x{\in}a \longrightarrow transitive\text{-}set(M,x))$

definition
limit-ordinal :: [*i*=>*o*,*i*] => *o* **where**
— a limit ordinal is a non-empty, successor-closed ordinal
$limit\text{-}ordinal(M,a) ==$
$ordinal(M,a)\ \&\ {\sim}\ empty(M,a)\ \&$
$(\forall\, x[M].\ x{\in}a \longrightarrow (\exists\, y[M].\ y{\in}a\ \&\ successor(M,x,y)))$

definition
successor-ordinal :: [*i*=>*o*,*i*] => *o* **where**
— a successor ordinal is any ordinal that is neither empty nor limit
$successor\text{-}ordinal(M,a) ==$
$ordinal(M,a)\ \&\ {\sim}\ empty(M,a)\ \&\ {\sim}\ limit\text{-}ordinal(M,a)$

definition
finite-ordinal :: [*i*=>*o*,*i*] => *o* **where**
— an ordinal is finite if neither it nor any of its elements are limit
finite-ordinal(*M*,*a*) ==
ordinal(*M*,*a*) & ~ *limit-ordinal*(*M*,*a*) &
(∀ *x*[*M*]. *x*∈*a* ⟶ ~ *limit-ordinal*(*M*,*x*))

definition
omega :: [*i*=>*o*,*i*] => *o* **where**
— omega is a limit ordinal none of whose elements are limit
omega(*M*,*a*) == *limit-ordinal*(*M*,*a*) & (∀ *x*[*M*]. *x*∈*a* ⟶ ~ *limit-ordinal*(*M*,*x*))

definition
is-quasinat :: [*i*=>*o*,*i*] => *o* **where**
is-quasinat(*M*,*z*) == *empty*(*M*,*z*) | (∃ *m*[*M*]. *successor*(*M*,*m*,*z*))

definition
is-nat-case :: [*i*=>*o*, *i*, [*i*,*i*]=>*o*, *i*, *i*] => *o* **where**
is-nat-case(*M*, *a*, *is-b*, *k*, *z*) ==
(*empty*(*M*,*k*) ⟶ *z*=*a*) &
(∀ *m*[*M*]. *successor*(*M*,*m*,*k*) ⟶ *is-b*(*m*,*z*)) &
(*is-quasinat*(*M*,*k*) | *empty*(*M*,*z*))

definition
relation1 :: [*i*=>*o*, [*i*,*i*]=>*o*, *i*=>*i*] => *o* **where**
relation1(*M*,*is-f*,*f*) == ∀ *x*[*M*]. ∀ *y*[*M*]. *is-f*(*x*,*y*) ⟷ *y* = *f*(*x*)

definition
Relation1 :: [*i*=>*o*, *i*, [*i*,*i*]=>*o*, *i*=>*i*] => *o* **where**
— as above, but typed
Relation1(*M*,*A*,*is-f*,*f*) ==
∀ *x*[*M*]. ∀ *y*[*M*]. *x*∈*A* ⟶ *is-f*(*x*,*y*) ⟷ *y* = *f*(*x*)

definition
relation2 :: [*i*=>*o*, [*i*,*i*,*i*]=>*o*, [*i*,*i*]=>*i*] => *o* **where**
relation2(*M*,*is-f*,*f*) == ∀ *x*[*M*]. ∀ *y*[*M*]. ∀ *z*[*M*]. *is-f*(*x*,*y*,*z*) ⟷ *z* = *f*(*x*,*y*)

definition
Relation2 :: [*i*=>*o*, *i*, *i*, [*i*,*i*,*i*]=>*o*, [*i*,*i*]=>*i*] => *o* **where**
Relation2(*M*,*A*,*B*,*is-f*,*f*) ==
∀ *x*[*M*]. ∀ *y*[*M*]. ∀ *z*[*M*]. *x*∈*A* ⟶ *y*∈*B* ⟶ *is-f*(*x*,*y*,*z*) ⟷ *z* = *f*(*x*,*y*)

definition
relation3 :: [*i*=>*o*, [*i*,*i*,*i*,*i*]=>*o*, [*i*,*i*,*i*]=>*i*] => *o* **where**
relation3(*M*,*is-f*,*f*) ==
∀ *x*[*M*]. ∀ *y*[*M*]. ∀ *z*[*M*]. ∀ *u*[*M*]. *is-f*(*x*,*y*,*z*,*u*) ⟷ *u* = *f*(*x*,*y*,*z*)

definition
Relation3 :: [*i*=>*o*, *i*, *i*, *i*, [*i*,*i*,*i*,*i*]=>*o*, [*i*,*i*,*i*]=>*i*] => *o* **where**

Relation3(*M*,*A*,*B*,*C*,*is-f*,*f*) ==
$\forall x[M]. \forall y[M]. \forall z[M]. \forall u[M].$
$x{\in}A \longrightarrow y{\in}B \longrightarrow z{\in}C \longrightarrow$ *is-f*$(x,y,z,u) \longleftrightarrow u = f(x,y,z)$

definition
relation4 :: [*i*=>*o*, [*i*,*i*,*i*,*i*,*i*]=>*o*, [*i*,*i*,*i*,*i*]=>*i*] => *o* **where**
relation4(*M*,*is-f*,*f*) ==
$\forall u[M]. \forall x[M]. \forall y[M]. \forall z[M]. \forall a[M].$ *is-f*$(u,x,y,z,a) \longleftrightarrow a = f(u,x,y,z)$

Useful when absoluteness reasoning has replaced the predicates by terms

lemma *triv-Relation1*:
Relation1$(M, A, \lambda x\ y.\ y = f(x), f)$
by (*simp add*: *Relation1-def*)

lemma *triv-Relation2*:
Relation2$(M, A, B, \lambda x\ y\ a.\ a = f(x,y), f)$
by (*simp add*: *Relation2-def*)

1.2 The relativized ZF axioms

definition
extensionality :: (*i*=>*o*) => *o* **where**
extensionality(*M*) ==
$\forall x[M]. \forall y[M]. (\forall z[M]. z \in x \longleftrightarrow z \in y) \longrightarrow x{=}y$

definition
separation :: [*i*=>*o*, *i*=>*o*] => *o* **where**
— The formula *P* should only involve parameters belonging to *M* and all its quantifiers must be relativized to *M*. We do not have separation as a scheme; every instance that we need must be assumed (and later proved) separately.
separation(*M*,*P*) ==
$\forall z[M]. \exists y[M]. \forall x[M]. x \in y \longleftrightarrow x \in z\ \&\ P(x)$

definition
upair-ax :: (*i*=>*o*) => *o* **where**
upair-ax(*M*) == $\forall x[M]. \forall y[M]. \exists z[M].$ *upair*(*M*,*x*,*y*,*z*)

definition
Union-ax :: (*i*=>*o*) => *o* **where**
Union-ax(*M*) == $\forall x[M]. \exists z[M].$ *big-union*(*M*,*x*,*z*)

definition
power-ax :: (*i*=>*o*) => *o* **where**
power-ax(*M*) == $\forall x[M]. \exists z[M].$ *powerset*(*M*,*x*,*z*)

definition
univalent :: [*i*=>*o*, *i*, [*i*,*i*]=>*o*] => *o* **where**
univalent(*M*,*A*,*P*) ==
$\forall x[M]. x{\in}A \longrightarrow (\forall y[M]. \forall z[M]. P(x,y)\ \&\ P(x,z) \longrightarrow y{=}z)$

definition
replacement :: [*i*=>*o*, [*i*,*i*]=>*o*] => *o* **where**
replacement(*M*,*P*) ==
∀ *A*[*M*]. *univalent*(*M*,*A*,*P*) ⟶
(∃ *Y*[*M*]. ∀ *b*[*M*]. (∃ *x*[*M*]. *x*∈*A* & *P*(*x*,*b*)) ⟶ *b* ∈ *Y*)

definition
strong-replacement :: [*i*=>*o*, [*i*,*i*]=>*o*] => *o* **where**
strong-replacement(*M*,*P*) ==
∀ *A*[*M*]. *univalent*(*M*,*A*,*P*) ⟶
(∃ *Y*[*M*]. ∀ *b*[*M*]. *b* ∈ *Y* ⟷ (∃ *x*[*M*]. *x*∈*A* & *P*(*x*,*b*)))

definition
foundation-ax :: (*i*=>*o*) => *o* **where**
foundation-ax(*M*) ==
∀ *x*[*M*]. (∃ *y*[*M*]. *y*∈*x*) ⟶ (∃ *y*[*M*]. *y*∈*x* & ~(∃ *z*[*M*]. *z*∈*x* & *z* ∈ *y*))

1.3 A trivial consistency proof for V_ω

We prove that V_ω (or *univ* in Isabelle) satisfies some ZF axioms. Kunen, Theorem IV 3.13, page 123.

lemma *univ0-downwards-mem*: [| *y* ∈ *x*; *x* ∈ *univ*(*0*) |] ==> *y* ∈ *univ*(*0*)
apply (*insert Transset-univ* [*OF Transset-0*])
apply (*simp add*: *Transset-def*, *blast*)
done

lemma *univ0-Ball-abs* [*simp*]:
A ∈ *univ*(*0*) ==> (∀ *x*∈*A*. *x* ∈ *univ*(*0*) ⟶ *P*(*x*)) ⟷ (∀ *x*∈*A*. *P*(*x*))
by (*blast intro*: *univ0-downwards-mem*)

lemma *univ0-Bex-abs* [*simp*]:
A ∈ *univ*(*0*) ==> (∃ *x*∈*A*. *x* ∈ *univ*(*0*) & *P*(*x*)) ⟷ (∃ *x*∈*A*. *P*(*x*))
by (*blast intro*: *univ0-downwards-mem*)

Congruence rule for separation: can assume the variable is in M

lemma *separation-cong* [*cong*]:
(!!*x*. *M*(*x*) ==> *P*(*x*) ⟷ *P'*(*x*))
==> *separation*(*M*, %*x*. *P*(*x*)) ⟷ *separation*(*M*, %*x*. *P'*(*x*))
by (*simp add*: *separation-def*)

lemma *univalent-cong* [*cong*]:
[| *A*=*A'*; !!*x y*. [| *x*∈*A*; *M*(*x*); *M*(*y*) |] ==> *P*(*x*,*y*) ⟷ *P'*(*x*,*y*) |]
==> *univalent*(*M*, *A*, %*x y*. *P*(*x*,*y*)) ⟷ *univalent*(*M*, *A'*, %*x y*. *P'*(*x*,*y*))
by (*simp add*: *univalent-def*)

lemma *univalent-triv* [*intro*,*simp*]:
univalent(*M*, *A*, λ*x y*. *y* = *f*(*x*))

by (*simp add*: *univalent-def*)

lemma *univalent-conjI2* [*intro*,*simp*]:
univalent(M,A,Q) ==> *univalent*(M, A, $\lambda x\ y.\ P(x,y)$ & $Q(x,y)$)
by (*simp add*: *univalent-def*, *blast*)

Congruence rule for replacement

lemma *strong-replacement-cong* [*cong*]:
[| !!$x\ y$. [| $M(x)$; $M(y)$ |] ==> $P(x,y) \longleftrightarrow P'(x,y)$ |]
==> *strong-replacement*(M, %$x\ y$. $P(x,y)$) $\longleftrightarrow$
strong-replacement(M, %$x\ y$. $P'(x,y)$)
by (*simp add*: *strong-replacement-def*)

The extensionality axiom

lemma *extensionality*($\lambda x.\ x \in$ *univ*(*0*))
apply (*simp add*: *extensionality-def*)
apply (*blast intro*: *univ0-downwards-mem*)
done

The separation axiom requires some lemmas

lemma *Collect-in-Vfrom*:
[| $X \in$ *Vfrom*(A,j); *Transset*(A) |] ==> *Collect*(X,P) $\in$ *Vfrom*(A, *succ*(j))
apply (*drule Transset-Vfrom*)
apply (*rule subset-mem-Vfrom*)
apply (*unfold Transset-def*, *blast*)
done

lemma *Collect-in-VLimit*:
[| $X \in$ *Vfrom*(A,i); *Limit*(i); *Transset*(A) |]
==> *Collect*(X,P) $\in$ *Vfrom*(A,i)
apply (*rule Limit-VfromE*, *assumption+*)
apply (*blast intro*: *Limit-has-succ VfromI Collect-in-Vfrom*)
done

lemma *Collect-in-univ*:
[| $X \in$ *univ*(A); *Transset*(A) |] ==> *Collect*(X,P) $\in$ *univ*(A)
by (*simp add*: *univ-def Collect-in-VLimit*)

lemma *separation*($\lambda x.\ x \in$ *univ*(*0*), P)
apply (*simp add*: *separation-def*, *clarify*)
apply (*rule-tac* x = *Collect*(z,P) **in** *bexI*)
apply (*blast intro*: *Collect-in-univ Transset-0*)+
done

Unordered pairing axiom

lemma *upair-ax*($\lambda x.\ x \in$ *univ*(*0*))
apply (*simp add*: *upair-ax-def upair-def*)
apply (*blast intro*: *doubleton-in-univ*)

done

Union axiom

lemma *Union-ax*$(\lambda x.\ x \in univ(0))$
apply (*simp add*: *Union-ax-def big-union-def*, *clarify*)
apply (*rule-tac* $x{=}\bigcup x$ **in** *bexI*)
 apply (*blast intro*: *univ0-downwards-mem*)
apply (*blast intro*: *Union-in-univ Transset-0*)
done

Powerset axiom

lemma *Pow-in-univ*:
    $[|\ X \in univ(A);\ \ Transset(A)\ |] ==> Pow(X) \in univ(A)$
apply (*simp add*: *univ-def Pow-in-VLimit*)
done

lemma *power-ax*$(\lambda x.\ x \in univ(0))$
apply (*simp add*: *power-ax-def powerset-def subset-def*, *clarify*)
apply (*rule-tac* $x{=}Pow(x)$ **in** *bexI*)
 apply (*blast intro*: *univ0-downwards-mem*)
apply (*blast intro*: *Pow-in-univ Transset-0*)
done

Foundation axiom

lemma *foundation-ax*$(\lambda x.\ x \in univ(0))$
apply (*simp add*: *foundation-ax-def*, *clarify*)
apply (*cut-tac* $A{=}x$ **in** *foundation*)
apply (*blast intro*: *univ0-downwards-mem*)
done

lemma *replacement*$(\lambda x.\ x \in univ(0),\ P)$
apply (*simp add*: *replacement-def*, *clarify*)
oops

no idea: maybe prove by induction on the rank of A?

Still missing: Replacement, Choice

1.4 Lemmas Needed to Reduce Some Set Constructions to Instances of Separation

lemma *image-iff-Collect*: $r\ ``\ A = \{y \in \bigcup(\bigcup(r)).\ \exists\, p{\in}r.\ \exists\, x{\in}A.\ p{=}<x,y>\}$
apply (*rule equalityI*, *auto*)
apply (*simp add*: *Pair-def*, *blast*)
done

lemma *vimage-iff-Collect*:
    $r\ -``\ A = \{x \in \bigcup(\bigcup(r)).\ \exists\, p{\in}r.\ \exists\, y{\in}A.\ p{=}<x,y>\}$
apply (*rule equalityI*, *auto*)

apply (*simp add*: *Pair-def*, *blast*)
done

These two lemmas lets us prove *domain-closed* and *range-closed* without new instances of separation

lemma *domain-eq-vimage*: *domain*(*r*) = *r* −‘‘ *Union*(*Union*(*r*))
apply (*rule equalityI*, *auto*)
apply (*rule vimageI*, *assumption*)
apply (*simp add*: *Pair-def*, *blast*)
done

lemma *range-eq-image*: *range*(*r*) = *r* ‘‘ *Union*(*Union*(*r*))
apply (*rule equalityI*, *auto*)
apply (*rule imageI*, *assumption*)
apply (*simp add*: *Pair-def*, *blast*)
done

lemma *replacementD*:
  [| *replacement*(*M*,*P*); *M*(*A*); *univalent*(*M*,*A*,*P*) |]
  ==> ∃ *Y*[*M*]. (∀ *b*[*M*]. ((∃ *x*[*M*]. *x*∈*A* & *P*(*x*,*b*)) ⟶ *b* ∈ *Y*))
by (*simp add*: *replacement-def*)

lemma *strong-replacementD*:
  [| *strong-replacement*(*M*,*P*); *M*(*A*); *univalent*(*M*,*A*,*P*) |]
  ==> ∃ *Y*[*M*]. (∀ *b*[*M*]. (*b* ∈ *Y* ⟷ (∃ *x*[*M*]. *x*∈*A* & *P*(*x*,*b*))))
by (*simp add*: *strong-replacement-def*)

lemma *separationD*:
  [| *separation*(*M*,*P*); *M*(*z*) |] ==> ∃ *y*[*M*]. ∀ *x*[*M*]. *x* ∈ *y* ⟷ *x* ∈ *z* & *P*(*x*)
by (*simp add*: *separation-def*)

More constants, for order types

definition
 order-isomorphism :: [*i*=>*o*,*i*,*i*,*i*,*i*,*i*] => *o* **where**
  order-isomorphism(*M*,*A*,*r*,*B*,*s*,*f*) ==
    bijection(*M*,*A*,*B*,*f*) &
    (∀ *x*[*M*]. *x*∈*A* ⟶ (∀ *y*[*M*]. *y*∈*A* ⟶
     (∀ *p*[*M*]. ∀ *fx*[*M*]. ∀ *fy*[*M*]. ∀ *q*[*M*].
      pair(*M*,*x*,*y*,*p*) ⟶ *fun-apply*(*M*,*f*,*x*,*fx*) ⟶ *fun-apply*(*M*,*f*,*y*,*fy*) ⟶
      pair(*M*,*fx*,*fy*,*q*) ⟶ (*p*∈*r* ⟷ *q*∈*s*))))

definition
 pred-set :: [*i*=>*o*,*i*,*i*,*i*,*i*] => *o* **where**
  pred-set(*M*,*A*,*x*,*r*,*B*) ==
    ∀ *y*[*M*]. *y* ∈ *B* ⟷ (∃ *p*[*M*]. *p*∈*r* & *y* ∈ *A* & *pair*(*M*,*y*,*x*,*p*))

definition
 membership :: [*i*=>*o*,*i*,*i*] => *o* **where** — membership relation
  membership(*M*,*A*,*r*) ==

$\forall\, p[M].\ p \in r \longleftrightarrow (\exists\, x[M].\ x{\in}A\ \&\ (\exists\, y[M].\ y{\in}A\ \&\ x{\in}y\ \&\ pair(M,x,y,p)))$

1.5 Introducing a Transitive Class Model

The class M is assumed to be transitive and to satisfy some relativized ZF axioms

locale *M-trivial* =
 fixes *M*
 assumes *transM*: [| $y{\in}x$; $M(x)$ |] ==> $M(y)$
  and *upair-ax*: *upair-ax*(*M*)
  and *Union-ax*: *Union-ax*(*M*)

  and *M-inhabit* [*iff*]: *M*(*0*)

Automatically discovers the proof using *transM*, *nat-0I* and *M-inhabit*.

lemma (**in** *M-trivial*) *rall-abs* [*simp*]:
 $M(A)$ ==> $(\forall\, x[M].\ x{\in}A \longrightarrow P(x)) \longleftrightarrow (\forall\, x{\in}A.\ P(x))$
by (*blast intro*: *transM*)

lemma (**in** *M-trivial*) *rex-abs* [*simp*]:
 $M(A)$ ==> $(\exists\, x[M].\ x{\in}A\ \&\ P(x)) \longleftrightarrow (\exists\, x{\in}A.\ P(x))$
by (*blast intro*: *transM*)

lemma (**in** *M-trivial*) *ball-iff-equiv*:
 $M(A)$ ==> $(\forall\, x[M].\ (x{\in}A \longleftrightarrow P(x))) \longleftrightarrow$
  $(\forall\, x{\in}A.\ P(x))\ \&\ (\forall\, x.\ P(x) \longrightarrow M(x) \longrightarrow x{\in}A)$
by (*blast intro*: *transM*)

Simplifies proofs of equalities when there's an iff-equality available for rewriting, universally quantified over M. But it's not the only way to prove such equalities: its premises $M(A)$ and $M(B)$ can be too strong.

lemma (**in** *M-trivial*) *M-equalityI*:
 [| !!*x*. $M(x)$ ==> $x{\in}A \longleftrightarrow x{\in}B$; $M(A)$; $M(B)$ |] ==> $A{=}B$
by (*blast dest*: *transM*)

1.5.1 Trivial Absoluteness Proofs: Empty Set, Pairs, etc.

lemma (**in** *M-trivial*) *empty-abs* [*simp*]:
 $M(z)$ ==> $empty(M,z) \longleftrightarrow z{=}0$
apply (*simp add*: *empty-def*)
apply (*blast intro*: *transM*)
done

lemma (**in** *M-trivial*) *subset-abs* [*simp*]:
 $M(A)$ ==> $subset(M,A,B) \longleftrightarrow A \subseteq B$
apply (*simp add*: *subset-def*)
apply (*blast intro*: *transM*)

done

lemma (**in** *M-trivial*) *upair-abs* [*simp*]:
  M(*z*) ==> *upair*(*M*,*a*,*b*,*z*) ⟷ *z*={*a*,*b*}
apply (*simp add*: *upair-def*)
apply (*blast intro*: *transM*)
done

lemma (**in** *M-trivial*) *upair-in-M-iff* [*iff*]:
  M({*a*,*b*}) ⟷ *M*(*a*) & *M*(*b*)
apply (*insert upair-ax*, *simp add*: *upair-ax-def*)
apply (*blast intro*: *transM*)
done

lemma (**in** *M-trivial*) *singleton-in-M-iff* [*iff*]:
  M({*a*}) ⟷ *M*(*a*)
by (*insert upair-in-M-iff* [*of a a*], *simp*)

lemma (**in** *M-trivial*) *pair-abs* [*simp*]:
  M(*z*) ==> *pair*(*M*,*a*,*b*,*z*) ⟷ *z*=<*a*,*b*>
apply (*simp add*: *pair-def Pair-def*)
apply (*blast intro*: *transM*)
done

lemma (**in** *M-trivial*) *pair-in-M-iff* [*iff*]:
  M(<*a*,*b*>) ⟷ *M*(*a*) & *M*(*b*)
by (*simp add*: *Pair-def*)

lemma (**in** *M-trivial*) *pair-components-in-M*:
  [| <*x*,*y*> ∈ *A*; *M*(*A*) |] ==> *M*(*x*) & *M*(*y*)
apply (*simp add*: *Pair-def*)
apply (*blast dest*: *transM*)
done

lemma (**in** *M-trivial*) *cartprod-abs* [*simp*]:
  [| *M*(*A*); *M*(*B*); *M*(*z*) |] ==> *cartprod*(*M*,*A*,*B*,*z*) ⟷ *z* = *A*∗*B*
apply (*simp add*: *cartprod-def*)
apply (*rule iffI*)
 apply (*blast intro*!: *equalityI intro*: *transM dest*!: *rspec*)
apply (*blast dest*: *transM*)
done

1.5.2 Absoluteness for Unions and Intersections

lemma (**in** *M-trivial*) *union-abs* [*simp*]:
  [| *M*(*a*); *M*(*b*); *M*(*z*) |] ==> *union*(*M*,*a*,*b*,*z*) ⟷ *z* = *a* ∪ *b*
apply (*simp add*: *union-def*)
apply (*blast intro*: *transM*)
done

lemma (**in** *M-trivial*) *inter-abs* [*simp*]:
[| $M(a)$; $M(b)$; $M(z)$ |] ==> $inter(M,a,b,z) \longleftrightarrow z = a \cap b$
apply (*simp add*: *inter-def*)
apply (*blast intro*: *transM*)
done

lemma (**in** *M-trivial*) *setdiff-abs* [*simp*]:
[| $M(a)$; $M(b)$; $M(z)$ |] ==> $setdiff(M,a,b,z) \longleftrightarrow z = a-b$
apply (*simp add*: *setdiff-def*)
apply (*blast intro*: *transM*)
done

lemma (**in** *M-trivial*) *Union-abs* [*simp*]:
[| $M(A)$; $M(z)$ |] ==> $big\text{-}union(M,A,z) \longleftrightarrow z = \bigcup(A)$
apply (*simp add*: *big-union-def*)
apply (*blast dest*: *transM*)
done

lemma (**in** *M-trivial*) *Union-closed* [*intro*,*simp*]:
$M(A)$ ==> $M(\bigcup(A))$
by (*insert Union-ax*, *simp add*: *Union-ax-def*)

lemma (**in** *M-trivial*) *Un-closed* [*intro*,*simp*]:
[| $M(A)$; $M(B)$ |] ==> $M(A \cup B)$
by (*simp only*: *Un-eq-Union*, *blast*)

lemma (**in** *M-trivial*) *cons-closed* [*intro*,*simp*]:
[| $M(a)$; $M(A)$ |] ==> $M(cons(a,A))$
by (*subst cons-eq* [*symmetric*], *blast*)

lemma (**in** *M-trivial*) *cons-abs* [*simp*]:
[| $M(b)$; $M(z)$ |] ==> $is\text{-}cons(M,a,b,z) \longleftrightarrow z = cons(a,b)$
by (*simp add*: *is-cons-def*, *blast intro*: *transM*)

lemma (**in** *M-trivial*) *successor-abs* [*simp*]:
[| $M(a)$; $M(z)$ |] ==> $successor(M,a,z) \longleftrightarrow z = succ(a)$
by (*simp add*: *successor-def*, *blast*)

lemma (**in** *M-trivial*) *succ-in-M-iff* [*iff*]:
$M(succ(a)) \longleftrightarrow M(a)$
apply (*simp add*: *succ-def*)
apply (*blast intro*: *transM*)
done

1.5.3 Absoluteness for Separation and Replacement

lemma (**in** *M-trivial*) *separation-closed* [*intro*,*simp*]:
[| $separation(M,P)$; $M(A)$ |] ==> $M(Collect(A,P))$

apply (*insert separation*, *simp add*: *separation-def*)
apply (*drule rspec*, *assumption*, *clarify*)
apply (*subgoal-tac y* = *Collect*(*A*,*P*), *blast*)
apply (*blast dest*: *transM*)
done

lemma *separation-iff*:
  separation(*M*,*P*) ⟷ (∀ *z*[*M*]. ∃ *y*[*M*]. *is-Collect*(*M*,*z*,*P*,*y*))
by (*simp add*: *separation-def is-Collect-def*)

lemma (**in** *M-trivial*) *Collect-abs* [*simp*]:
  [| *M*(*A*); *M*(*z*) |] ==> *is-Collect*(*M*,*A*,*P*,*z*) ⟷ *z* = *Collect*(*A*,*P*)
apply (*simp add*: *is-Collect-def*)
apply (*blast dest*: *transM*)
done

Probably the premise and conclusion are equivalent

1.5.4 The Operator *is-Replace*

lemma *is-Replace-cong* [*cong*]:
  [| *A*=*A*′;
    !!*x y*. [| *M*(*x*); *M*(*y*) |] ==> *P*(*x*,*y*) ⟷ *P*′(*x*,*y*);
    z=*z*′ |]
  ==> *is-Replace*(*M*, *A*, %*x y*. *P*(*x*,*y*), *z*) ⟷
    is-Replace(*M*, *A*′, %*x y*. *P*′(*x*,*y*), *z*′)
by (*simp add*: *is-Replace-def*)

lemma (**in** *M-trivial*) *univalent-Replace-iff*:
  [| *M*(*A*); *univalent*(*M*,*A*,*P*);
    !!*x y*. [| *x*∈*A*; *P*(*x*,*y*) |] ==> *M*(*y*) |]
  ==> *u* ∈ *Replace*(*A*,*P*) ⟷ (∃ *x*. *x*∈*A* & *P*(*x*,*u*))
apply (*simp add*: *Replace-iff univalent-def*)
apply (*blast dest*: *transM*)
done

lemma (**in** *M-trivial*) *strong-replacement-closed* [*intro*,*simp*]:
  [| *strong-replacement*(*M*,*P*); *M*(*A*); *univalent*(*M*,*A*,*P*);
    !!*x y*. [| *x*∈*A*; *P*(*x*,*y*) |] ==> *M*(*y*) |] ==> *M*(*Replace*(*A*,*P*))
apply (*simp add*: *strong-replacement-def*)
apply (*drule-tac x*=*A* **in** *rspec*, *safe*)
apply (*subgoal-tac Replace*(*A*,*P*) = *Y*)
 apply *simp*
apply (*rule equality-iffI*)
apply (*simp add*: *univalent-Replace-iff*)
apply (*blast dest*: *transM*)
done

lemma (**in** *M-trivial*) *Replace-abs*:
  [| *M*(*A*); *M*(*z*); *univalent*(*M*,*A*,*P*);
    !!*x y*. [| *x*∈*A*; *P*(*x*,*y*) |] ==> *M*(*y*) |]
   ==> *is-Replace*(*M*,*A*,*P*,*z*) ⟷ *z* = *Replace*(*A*,*P*)
apply (*simp add*: *is-Replace-def*)
apply (*rule iffI*)
 apply (*rule equality-iffI*)
 apply (*simp-all add*: *univalent-Replace-iff*)
 apply (*blast dest*: *transM*)+
done

lemma (**in** *M-trivial*) *RepFun-closed*:
  [| *strong-replacement*(*M*, λ*x y*. *y* = *f*(*x*)); *M*(*A*); ∀*x*∈*A*. *M*(*f*(*x*)) |]
   ==> *M*(*RepFun*(*A*,*f*))
apply (*simp add*: *RepFun-def*)
done

lemma *Replace-conj-eq*: {*y* . *x* ∈ *A*, *x*∈*A* & *y*=*f*(*x*)} = {*y* . *x*∈*A*, *y*=*f*(*x*)}
by *simp*

Better than *RepFun-closed* when having the formula $x \in A$ makes relativization easier.

lemma (**in** *M-trivial*) *RepFun-closed2*:
  [| *strong-replacement*(*M*, λ*x y*. *x*∈*A* & *y* = *f*(*x*)); *M*(*A*); ∀*x*∈*A*. *M*(*f*(*x*)) |]
   ==> *M*(*RepFun*(*A*, %*x*. *f*(*x*)))
apply (*simp add*: *RepFun-def*)
apply (*frule strong-replacement-closed*, *assumption*)
apply (*auto dest*: *transM* *simp add*: *Replace-conj-eq univalent-def*)
done

1.5.5 Absoluteness for *Lambda*

definition
 is-lambda :: [*i*=>*o*, *i*, [*i*,*i*]=>*o*, *i*] => *o* **where**
  is-lambda(*M*, *A*, *is-b*, *z*) ==
   ∀*p*[*M*]. *p* ∈ *z* ⟷
    (∃*u*[*M*]. ∃*v*[*M*]. *u*∈*A* & *pair*(*M*,*u*,*v*,*p*) & *is-b*(*u*,*v*))

lemma (**in** *M-trivial*) *lam-closed*:
  [| *strong-replacement*(*M*, λ*x y*. *y* = <*x*,*b*(*x*)>); *M*(*A*); ∀*x*∈*A*. *M*(*b*(*x*)) |]
   ==> *M*(λ*x*∈*A*. *b*(*x*))
by (*simp add*: *lam-def*, *blast intro*: *RepFun-closed dest*: *transM*)

Better than *lam-closed*: has the formula $x \in A$

lemma (**in** *M-trivial*) *lam-closed2*:
 [|*strong-replacement*(*M*, λ*x y*. *x*∈*A* & *y* = ⟨*x*, *b*(*x*)⟩);
  M(*A*); ∀*m*[*M*]. *m*∈*A* ⟶ *M*(*b*(*m*))|] ==> *M*(*Lambda*(*A*,*b*))

apply (*simp add*: *lam-def*)
apply (*blast intro*: *RepFun-closed2 dest*: *transM*)
done

lemma (**in** *M-trivial*) *lambda-abs2*:
  [| *Relation1*(*M*,*A*,*is-b*,*b*); *M*(*A*); ∀ *m*[*M*]. *m*∈*A* ⟶ *M*(*b*(*m*)); *M*(*z*) |]
  ==> *is-lambda*(*M*,*A*,*is-b*,*z*) ⟷ *z* = *Lambda*(*A*,*b*)
apply (*simp add*: *Relation1-def is-lambda-def*)
apply (*rule iffI*)
 prefer *2* **apply** (*simp add*: *lam-def*)
apply (*rule equality-iffI*)
apply (*simp add*: *lam-def*)
apply (*rule iffI*)
 apply (*blast dest*: *transM*)
apply (*auto simp add*: *transM* [*of - A*])
done

lemma *is-lambda-cong* [*cong*]:
  [| *A*=*A*′; *z*=*z*′;
    !!*x y*. [| *x*∈*A*; *M*(*x*); *M*(*y*) |] ==> *is-b*(*x*,*y*) ⟷ *is-b*′(*x*,*y*) |]
  ==> *is-lambda*(*M*, *A*, %*x y*. *is-b*(*x*,*y*), *z*) ⟷
    is-lambda(*M*, *A*′, %*x y*. *is-b*′(*x*,*y*), *z*′)
by (*simp add*: *is-lambda-def*)

lemma (**in** *M-trivial*) *image-abs* [*simp*]:
  [| *M*(*r*); *M*(*A*); *M*(*z*) |] ==> *image*(*M*,*r*,*A*,*z*) ⟷ *z* = *r*‘‘*A*
apply (*simp add*: *image-def*)
apply (*rule iffI*)
 apply (*blast intro*!: *equalityI dest*: *transM*, *blast*)
done

What about *Pow-abs*? Powerset is NOT absolute! This result is one direction of absoluteness.

lemma (**in** *M-trivial*) *powerset-Pow*:
  powerset(*M*, *x*, *Pow*(*x*))
by (*simp add*: *powerset-def*)

But we can't prove that the powerset in *M* includes the real powerset.

lemma (**in** *M-trivial*) *powerset-imp-subset-Pow*:
  [| *powerset*(*M*,*x*,*y*); *M*(*y*) |] ==> *y* ⊆ *Pow*(*x*)
apply (*simp add*: *powerset-def*)
apply (*blast dest*: *transM*)
done

1.5.6 Absoluteness for the Natural Numbers

lemma (**in** *M-trivial*) *nat-into-M* [*intro*]:
  n ∈ *nat* ==> *M*(*n*)
by (*induct n rule*: *nat-induct*, *simp-all*)

```
lemma (in M-trivial) nat-case-closed [intro,simp]:
  [|M(k); M(a); ∀ m[M]. M(b(m))|] ==> M(nat-case(a,b,k))
apply (case-tac k=0, simp)
apply (case-tac ∃ m. k = succ(m), force)
apply (simp add: nat-case-def)
done
```

```
lemma (in M-trivial) quasinat-abs [simp]:
     M(z) ==> is-quasinat(M,z) ⟷ quasinat(z)
by (auto simp add: is-quasinat-def quasinat-def)
```

```
lemma (in M-trivial) nat-case-abs [simp]:
     [| relation1(M,is-b,b); M(k); M(z) |]
      ==> is-nat-case(M,a,is-b,k,z) ⟷ z = nat-case(a,b,k)
apply (case-tac quasinat(k))
 prefer 2
 apply (simp add: is-nat-case-def non-nat-case)
 apply (force simp add: quasinat-def)
apply (simp add: quasinat-def is-nat-case-def)
apply (elim disjE exE)
 apply (simp-all add: relation1-def)
done
```

```
lemma is-nat-case-cong:
     [| a = a'; k = k';  z = z';  M(z');
       !!x y. [| M(x); M(y) |] ==> is-b(x,y) ⟷ is-b'(x,y) |]
      ==> is-nat-case(M, a, is-b, k, z) ⟷ is-nat-case(M, a', is-b', k', z')
by (simp add: is-nat-case-def)
```

1.6 Absoluteness for Ordinals

These results constitute Theorem IV 5.1 of Kunen (page 126).

```
lemma (in M-trivial) lt-closed:
     [| j<i; M(i) |] ==> M(j)
by (blast dest: ltD intro: transM)
```

```
lemma (in M-trivial) transitive-set-abs [simp]:
     M(a) ==> transitive-set(M,a) ⟷ Transset(a)
by (simp add: transitive-set-def Transset-def)
```

```
lemma (in M-trivial) ordinal-abs [simp]:
     M(a) ==> ordinal(M,a) ⟷ Ord(a)
by (simp add: ordinal-def Ord-def)
```

```
lemma (in M-trivial) limit-ordinal-abs [simp]:
     M(a) ==> limit-ordinal(M,a) ⟷ Limit(a)
apply (unfold Limit-def limit-ordinal-def)
```

apply (*simp add*: *Ord-0-lt-iff*)
apply (*simp add*: *lt-def*, *blast*)
done

lemma (**in** *M-trivial*) *successor-ordinal-abs* [*simp*]:
  $M(a) ==> successor\text{-}ordinal(M,a) \longleftrightarrow Ord(a)\ \&\ (\exists\, b[M].\ a = succ(b))$
apply (*simp add*: *successor-ordinal-def*, *safe*)
apply (*drule Ord-cases-disj*, *auto*)
done

lemma *finite-Ord-is-nat*:
  $[|\ Ord(a);\ \sim\ Limit(a);\ \forall\, x{\in}a.\ \sim\ Limit(x)\ |] ==> a \in nat$
by (*induct a rule*: *trans-induct3*, *simp-all*)

lemma (**in** *M-trivial*) *finite-ordinal-abs* [*simp*]:
  $M(a) ==> finite\text{-}ordinal(M,a) \longleftrightarrow a \in nat$
apply (*simp add*: *finite-ordinal-def*)
apply (*blast intro*: *finite-Ord-is-nat intro*: *nat-into-Ord*
      dest: *Ord-trans naturals-not-limit*)
done

lemma *Limit-non-Limit-implies-nat*:
  $[|\ Limit(a);\ \forall\, x{\in}a.\ \sim\ Limit(x)\ |] ==> a = nat$
apply (*rule le-anti-sym*)
apply (*rule all-lt-imp-le*, *blast*, *blast intro*: *Limit-is-Ord*)
 apply (*simp add*: *lt-def*)
 apply (*blast intro*: *Ord-in-Ord Ord-trans finite-Ord-is-nat*)
apply (*erule nat-le-Limit*)
done

lemma (**in** *M-trivial*) *omega-abs* [*simp*]:
  $M(a) ==> omega(M,a) \longleftrightarrow a = nat$
apply (*simp add*: *omega-def*)
apply (*blast intro*: *Limit-non-Limit-implies-nat dest*: *naturals-not-limit*)
done

lemma (**in** *M-trivial*) *number1-abs* [*simp*]:
  $M(a) ==> number1(M,a) \longleftrightarrow a = 1$
by (*simp add*: *number1-def*)

lemma (**in** *M-trivial*) *number2-abs* [*simp*]:
  $M(a) ==> number2(M,a) \longleftrightarrow a = succ(1)$
by (*simp add*: *number2-def*)

lemma (**in** *M-trivial*) *number3-abs* [*simp*]:
  $M(a) ==> number3(M,a) \longleftrightarrow a = succ(succ(1))$
by (*simp add*: *number3-def*)

Kunen continued to 20...

1.7 Some instances of separation and strong replacement

```
locale M-basic = M-trivial +
assumes Inter-separation:
    M(A) ==> separation(M, λx. ∀y[M]. y∈A ⟶ x∈y)
  and Diff-separation:
    M(B) ==> separation(M, λx. x ∉ B)
  and cartprod-separation:
    [| M(A); M(B) |]
     ==> separation(M, λz. ∃x[M]. x∈A & (∃y[M]. y∈B & pair(M,x,y,z)))
  and image-separation:
    [| M(A); M(r) |]
     ==> separation(M, λy. ∃p[M]. p∈r & (∃x[M]. x∈A & pair(M,x,y,p)))
  and converse-separation:
    M(r) ==> separation(M,
       λz. ∃p[M]. p∈r & (∃x[M]. ∃y[M]. pair(M,x,y,p) & pair(M,y,x,z)))
  and restrict-separation:
    M(A) ==> separation(M, λz. ∃x[M]. x∈A & (∃y[M]. pair(M,x,y,z)))
  and comp-separation:
    [| M(r); M(s) |]
     ==> separation(M, λxz. ∃x[M]. ∃y[M]. ∃z[M]. ∃xy[M]. ∃yz[M].
                pair(M,x,z,xz) & pair(M,x,y,xy) & pair(M,y,z,yz) &
                xy∈s & yz∈r)
  and pred-separation:
    [| M(r); M(x) |] ==> separation(M, λy. ∃p[M]. p∈r & pair(M,y,x,p))
  and Memrel-separation:
    separation(M, λz. ∃x[M]. ∃y[M]. pair(M,x,y,z) & x ∈ y)
  and funspace-succ-replacement:
    M(n) ==>
     strong-replacement(M, λp z. ∃f[M]. ∃b[M]. ∃nb[M]. ∃cnbf[M].
               pair(M,f,b,p) & pair(M,n,b,nb) & is-cons(M,nb,f,cnbf) &
               upair(M,cnbf,cnbf,z))
  and is-recfun-separation:
    — for well-founded recursion: used to prove is-recfun-equal
    [| M(r); M(f); M(g); M(a); M(b) |]
    ==> separation(M,
          λx. ∃xa[M]. ∃xb[M].
               pair(M,x,a,xa) & xa ∈ r & pair(M,x,b,xb) & xb ∈ r &
               (∃fx[M]. ∃gx[M]. fun-apply(M,f,x,fx) & fun-apply(M,g,x,gx) &
                                   fx ≠ gx))
    and power-ax:          power-ax(M)

lemma (in M-basic) cartprod-iff-lemma:
    [| M(C); ∀u[M]. u ∈ C ⟷ (∃x∈A. ∃y∈B. u = {{x}, {x,y}});
       powerset(M, A ∪ B, p1); powerset(M, p1, p2);  M(p2) |]
     ==> C = {u ∈ p2 . ∃x∈A. ∃y∈B. u = {{x}, {x,y}}}
apply (simp add: powerset-def)
apply (rule equalityI, clarify, simp)
 apply (frule transM, assumption)
 apply (frule transM, assumption, simp (no-asm-simp))
```

apply *blast*
apply *clarify*
apply (*frule transM*, *assumption*, *force*)
done

lemma (**in** *M-basic*) *cartprod-iff*:
  [| *M*(*A*); *M*(*B*); *M*(*C*) |]
  ==> *cartprod*(*M*,*A*,*B*,*C*) ⟷
   (∃ *p1*[*M*]. ∃ *p2*[*M*]. *powerset*(*M*,*A* ∪ *B*,*p1*) & *powerset*(*M*,*p1*,*p2*) &
     C = {*z* ∈ *p2*. ∃ *x*∈*A*. ∃ *y*∈*B*. *z* = <*x*,*y*>})
apply (*simp add*: *Pair-def cartprod-def*, *safe*)
defer *1*
 apply (*simp add*: *powerset-def*)
apply *blast*

Final, difficult case: the left-to-right direction of the theorem.

apply (*insert power-ax*, *simp add*: *power-ax-def*)
apply (*frule-tac x*=*A* ∪ *B* **and** *P*=λ*x*. *rex*(*M*,*Q*(*x*)) **for** *Q* **in** *rspec*)
apply (*blast*, *clarify*)
apply (*drule-tac x*=*z* **and** *P*=λ*x*. *rex*(*M*,*Q*(*x*)) **for** *Q* **in** *rspec*)
apply *assumption*
apply (*blast intro*: *cartprod-iff-lemma*)
done

lemma (**in** *M-basic*) *cartprod-closed-lemma*:
  [| *M*(*A*); *M*(*B*) |] ==> ∃ *C*[*M*]. *cartprod*(*M*,*A*,*B*,*C*)
apply (*simp del*: *cartprod-abs add*: *cartprod-iff*)
apply (*insert power-ax*, *simp add*: *power-ax-def*)
apply (*frule-tac x*=*A* ∪ *B* **and** *P*=λ*x*. *rex*(*M*,*Q*(*x*)) **for** *Q* **in** *rspec*)
apply (*blast*, *clarify*)
apply (*drule-tac x*=*z* **and** *P*=λ*x*. *rex*(*M*,*Q*(*x*)) **for** *Q* **in** *rspec*, *auto*)
apply (*intro rexI conjI*, *simp*+)
apply (*insert cartprod-separation* [*of A B*], *simp*)
done

All the lemmas above are necessary because Powerset is not absolute. I should have used Replacement instead!

lemma (**in** *M-basic*) *cartprod-closed* [*intro*,*simp*]:
  [| *M*(*A*); *M*(*B*) |] ==> *M*(*A*∗*B*)
by (*frule cartprod-closed-lemma*, *assumption*, *force*)

lemma (**in** *M-basic*) *sum-closed* [*intro*,*simp*]:
  [| *M*(*A*); *M*(*B*) |] ==> *M*(*A*+*B*)
by (*simp add*: *sum-def*)

lemma (**in** *M-basic*) *sum-abs* [*simp*]:
  [| *M*(*A*); *M*(*B*); *M*(*Z*) |] ==> *is-sum*(*M*,*A*,*B*,*Z*) ⟷ (*Z* = *A*+*B*)
by (*simp add*: *is-sum-def sum-def singleton-0 nat-into-M*)

lemma (**in** *M-trivial*) *Inl-in-M-iff* [*iff*]:
$M(Inl(a)) \longleftrightarrow M(a)$
by (*simp add*: *Inl-def*)

lemma (**in** *M-trivial*) *Inl-abs* [*simp*]:
$M(Z)$ ==> $is\text{-}Inl(M,a,Z) \longleftrightarrow (Z = Inl(a))$
by (*simp add*: *is-Inl-def Inl-def*)

lemma (**in** *M-trivial*) *Inr-in-M-iff* [*iff*]:
$M(Inr(a)) \longleftrightarrow M(a)$
by (*simp add*: *Inr-def*)

lemma (**in** *M-trivial*) *Inr-abs* [*simp*]:
$M(Z)$ ==> $is\text{-}Inr(M,a,Z) \longleftrightarrow (Z = Inr(a))$
by (*simp add*: *is-Inr-def Inr-def*)

1.7.1 converse of a relation

lemma (**in** *M-basic*) *M-converse-iff*:
$M(r)$ ==>
$converse(r) =$
$\{z \in \bigcup(\bigcup(r)) * \bigcup(\bigcup(r)).$
$\exists p \in r.\ \exists x[M].\ \exists y[M].\ p = \langle x,y \rangle\ \&\ z = \langle y,x \rangle\}$
apply (*rule equalityI*)
prefer *2* **apply** (*blast dest*: *transM*, *clarify*, *simp*)
apply (*simp add*: *Pair-def*)
apply (*blast dest*: *transM*)
done

lemma (**in** *M-basic*) *converse-closed* [*intro*,*simp*]:
$M(r)$ ==> $M(converse(r))$
apply (*simp add*: *M-converse-iff*)
apply (*insert converse-separation* [*of r*], *simp*)
done

lemma (**in** *M-basic*) *converse-abs* [*simp*]:
[| $M(r)$; $M(z)$ |] ==> $is\text{-}converse(M,r,z) \longleftrightarrow z = converse(r)$
apply (*simp add*: *is-converse-def*)
apply (*rule iffI*)
prefer *2* **apply** *blast*
apply (*rule M-equalityI*)
apply *simp*
apply (*blast dest*: *transM*)+
done

1.7.2 image, preimage, domain, range

lemma (**in** *M-basic*) *image-closed* [*intro*,*simp*]:
[| $M(A)$; $M(r)$ |] ==> $M(r``A)$
apply (*simp add*: *image-iff-Collect*)

apply (*insert image-separation* [*of A r*], *simp*)
done

lemma (**in** *M-basic*) *vimage-abs* [*simp*]:
[| $M(r)$; $M(A)$; $M(z)$ |] ==> *pre-image*$(M,r,A,z) \longleftrightarrow z = r-``A$
apply (*simp add*: *pre-image-def*)
apply (*rule iffI*)
apply (*blast intro*!: *equalityI dest*: *transM*, *blast*)
done

lemma (**in** *M-basic*) *vimage-closed* [*intro*,*simp*]:
[| $M(A)$; $M(r)$ |] ==> $M(r-``A)$
by (*simp add*: *vimage-def*)

1.7.3 Domain, range and field

lemma (**in** *M-basic*) *domain-abs* [*simp*]:
[| $M(r)$; $M(z)$ |] ==> *is-domain*$(M,r,z) \longleftrightarrow z =$ *domain*(r)
apply (*simp add*: *is-domain-def*)
apply (*blast intro*!: *equalityI dest*: *transM*)
done

lemma (**in** *M-basic*) *domain-closed* [*intro*,*simp*]:
$M(r)$ ==> $M($*domain*$(r))$
apply (*simp add*: *domain-eq-vimage*)
done

lemma (**in** *M-basic*) *range-abs* [*simp*]:
[| $M(r)$; $M(z)$ |] ==> *is-range*$(M,r,z) \longleftrightarrow z =$ *range*(r)
apply (*simp add*: *is-range-def*)
apply (*blast intro*!: *equalityI dest*: *transM*)
done

lemma (**in** *M-basic*) *range-closed* [*intro*,*simp*]:
$M(r)$ ==> $M($*range*$(r))$
apply (*simp add*: *range-eq-image*)
done

lemma (**in** *M-basic*) *field-abs* [*simp*]:
[| $M(r)$; $M(z)$ |] ==> *is-field*$(M,r,z) \longleftrightarrow z =$ *field*(r)
by (*simp add*: *is-field-def field-def*)

lemma (**in** *M-basic*) *field-closed* [*intro*,*simp*]:
$M(r)$ ==> $M($*field*$(r))$
by (*simp add*: *field-def*)

1.7.4 Relations, functions and application

lemma (**in** *M-basic*) *relation-abs* [*simp*]:
$M(r)$ ==> *is-relation*$(M,r) \longleftrightarrow$ *relation*(r)

apply (*simp add*: *is-relation-def relation-def*)
apply (*blast dest*!: *bspec dest*: *pair-components-in-M*)+
done

lemma (**in** *M-basic*) *function-abs* [*simp*]:
$M(r) ==> is\text{-}function(M,r) \longleftrightarrow function(r)$
apply (*simp add*: *is-function-def function-def*, *safe*)
apply (*frule transM*, *assumption*)
apply (*blast dest*: *pair-components-in-M*)+
done

lemma (**in** *M-basic*) *apply-closed* [*intro*,*simp*]:
$[|M(f); M(a)|] ==> M(f`a)$
by (*simp add*: *apply-def*)

lemma (**in** *M-basic*) *apply-abs* [*simp*]:
$[| M(f); M(x); M(y) |] ==> fun\text{-}apply(M,f,x,y) \longleftrightarrow f`x = y$
apply (*simp add*: *fun-apply-def apply-def*, *blast*)
done

lemma (**in** *M-basic*) *typed-function-abs* [*simp*]:
$[| M(A); M(f) |] ==> typed\text{-}function(M,A,B,f) \longleftrightarrow f \in A -> B$
apply (*auto simp add*: *typed-function-def relation-def Pi-iff*)
apply (*blast dest*: *pair-components-in-M*)+
done

lemma (**in** *M-basic*) *injection-abs* [*simp*]:
$[| M(A); M(f) |] ==> injection(M,A,B,f) \longleftrightarrow f \in inj(A,B)$
apply (*simp add*: *injection-def apply-iff inj-def*)
apply (*blast dest*: *transM* [*of - A*])
done

lemma (**in** *M-basic*) *surjection-abs* [*simp*]:
$[| M(A); M(B); M(f) |] ==> surjection(M,A,B,f) \longleftrightarrow f \in surj(A,B)$
by (*simp add*: *surjection-def surj-def*)

lemma (**in** *M-basic*) *bijection-abs* [*simp*]:
$[| M(A); M(B); M(f) |] ==> bijection(M,A,B,f) \longleftrightarrow f \in bij(A,B)$
by (*simp add*: *bijection-def bij-def*)

1.7.5 Composition of relations

lemma (**in** *M-basic*) *M-comp-iff*:
$[| M(r); M(s) |]$
$==> r\ O\ s =$
$\{xz \in domain(s) * range(r).$
$\exists x[M]. \exists y[M]. \exists z[M]. xz = \langle x,z\rangle \ \&\ \langle x,y\rangle \in s \ \&\ \langle y,z\rangle \in r\}$
apply (*simp add*: *comp-def*)
apply (*rule equalityI*)

apply *clarify*
apply *simp*
apply (*blast dest*: *transM*)+
done

lemma (**in** *M-basic*) *comp-closed* [*intro*,*simp*]:
[| $M(r)$; $M(s)$ |] ==> $M(r\ O\ s)$
apply (*simp add*: *M-comp-iff*)
apply (*insert comp-separation* [*of r s*], *simp*)
done

lemma (**in** *M-basic*) *composition-abs* [*simp*]:
[| $M(r)$; $M(s)$; $M(t)$ |] ==> *composition*$(M,r,s,t) \longleftrightarrow t = r\ O\ s$
apply *safe*

Proving *composition*$(M,\ r,\ s,\ r\ O\ s)$

prefer *2*
apply (*simp add*: *composition-def comp-def*)
apply (*blast dest*: *transM*)

Opposite implication

apply (*rule M-equalityI*)
apply (*simp add*: *composition-def comp-def*)
apply (*blast del*: *allE dest*: *transM*)+
done

no longer needed

lemma (**in** *M-basic*) *restriction-is-function*:
[| *restriction*(M,f,A,z); *function*(f); $M(f)$; $M(A)$; $M(z)$ |]
==> *function*(z)
apply (*simp add*: *restriction-def ball-iff-equiv*)
apply (*unfold function-def*, *blast*)
done

lemma (**in** *M-basic*) *restriction-abs* [*simp*]:
[| $M(f)$; $M(A)$; $M(z)$ |]
==> *restriction*$(M,f,A,z) \longleftrightarrow z =$ *restrict*(f,A)
apply (*simp add*: *ball-iff-equiv restriction-def restrict-def*)
apply (*blast intro*!: *equalityI dest*: *transM*)
done

lemma (**in** *M-basic*) *M-restrict-iff*:
$M(r)$ ==> *restrict*$(r,A) = \{z \in r\ .\ \exists x{\in}A.\ \exists y[M].\ z = \langle x,\ y\rangle\}$
by (*simp add*: *restrict-def*, *blast dest*: *transM*)

lemma (**in** *M-basic*) *restrict-closed* [*intro*,*simp*]:
[| $M(A)$; $M(r)$ |] ==> $M($*restrict*$(r,A))$
apply (*simp add*: *M-restrict-iff*)

apply (*insert restrict-separation* [*of A*], *simp*)
done

lemma (**in** *M-basic*) *Inter-abs* [*simp*]:
  [| $M(A)$; $M(z)$ |] ==> $big\text{-}inter(M,A,z) \longleftrightarrow z = \bigcap(A)$
apply (*simp add*: *big-inter-def Inter-def*)
apply (*blast intro*!: *equalityI dest*: *transM*)
done

lemma (**in** *M-basic*) *Inter-closed* [*intro*,*simp*]:
  $M(A)$ ==> $M(\bigcap(A))$
by (*insert Inter-separation*, *simp add*: *Inter-def*)

lemma (**in** *M-basic*) *Int-closed* [*intro*,*simp*]:
  [| $M(A)$; $M(B)$ |] ==> $M(A \cap B)$
apply (*subgoal-tac* $M(\{A,B\})$)
apply (*frule Inter-closed*, *force*+)
done

lemma (**in** *M-basic*) *Diff-closed* [*intro*,*simp*]:
  [|$M(A)$; $M(B)$|] ==> $M(A-B)$
by (*insert Diff-separation*, *simp add*: *Diff-def*)

1.7.6 Some Facts About Separation Axioms

lemma (**in** *M-basic*) *separation-conj*:
  [|*separation*(M,P); *separation*(M,Q)|] ==> *separation*$(M, \lambda z.\ P(z)\ \&\ Q(z))$
by (*simp del*: *separation-closed*
    add: *separation-iff Collect-Int-Collect-eq* [*symmetric*])

lemma *Collect-Un-Collect-eq*:
  Collect$(A,P) \cup$ *Collect*(A,Q) = *Collect*$(A, \%x.\ P(x) \mid Q(x))$
by *blast*

lemma *Diff-Collect-eq*:
  $A -$ *Collect*(A,P) = *Collect*$(A, \%x.\ {\sim}\ P(x))$
by *blast*

lemma (**in** *M-trivial*) *Collect-rall-eq*:
  $M(Y)$ ==> *Collect*$(A, \%x.\ \forall y[M].\ y \in Y \longrightarrow P(x,y))$ =
    (*if* $Y=0$ *then* A *else* $(\bigcap y \in Y.\ \{x \in A.\ P(x,y)\})$)
apply *simp*
apply (*blast dest*: *transM*)
done

lemma (**in** *M-basic*) *separation-disj*:
  [|*separation*(M,P); *separation*(M,Q)|] ==> *separation*$(M, \lambda z.\ P(z) \mid Q(z))$
by (*simp del*: *separation-closed*

add: *separation-iff Collect-Un-Collect-eq* [*symmetric*])

lemma (**in** *M-basic*) *separation-neg*:
separation(*M*,*P*) ==> *separation*(*M*, λ*z*. ~*P*(*z*))
by (*simp del*: *separation-closed*
add: *separation-iff Diff-Collect-eq* [*symmetric*])

lemma (**in** *M-basic*) *separation-imp*:
[|*separation*(*M*,*P*); *separation*(*M*,*Q*)|]
==> *separation*(*M*, λ*z*. *P*(*z*) ⟶ *Q*(*z*))
by (*simp add*: *separation-neg separation-disj not-disj-iff-imp* [*symmetric*])

This result is a hint of how little can be done without the Reflection Theorem. The quantifier has to be bounded by a set. We also need another instance of Separation!

lemma (**in** *M-basic*) *separation-rall*:
[|*M*(*Y*); ∀ *y*[*M*]. *separation*(*M*, λ*x*. *P*(*x*,*y*));
∀ *z*[*M*]. *strong-replacement*(*M*, λ*x y*. *y* = {*u* ∈ *z* . *P*(*u*,*x*)})|]
==> *separation*(*M*, λ*x*. ∀ *y*[*M*]. *y*∈*Y* ⟶ *P*(*x*,*y*))
apply (*simp del*: *separation-closed rall-abs*
add: *separation-iff Collect-rall-eq*)
apply (*blast intro*!: *RepFun-closed dest*: *transM*)
done

1.7.7 Functions and function space

The assumption *M*(*A* −> *B*) is unusual, but essential: in all but trivial cases, A-¿B cannot be expected to belong to *M*.

lemma (**in** *M-basic*) *is-funspace-abs* [*simp*]:
[|*M*(*A*); *M*(*B*); *M*(*F*); *M*(*A*−>*B*)|] ==> *is-funspace*(*M*,*A*,*B*,*F*) ⟷ *F* = *A*−>*B*
apply (*simp add*: *is-funspace-def*)
apply (*rule iffI*)
prefer *2* **apply** *blast*
apply (*rule M-equalityI*)
apply *simp-all*
done

lemma (**in** *M-basic*) *succ-fun-eq2*:
[|*M*(*B*); *M*(*n*−>*B*)|] ==>
succ(*n*) −> *B* =
⋃{*z*. *p* ∈ (*n*−>*B*)∗*B*, ∃ *f*[*M*]. ∃ *b*[*M*]. *p* = <*f*,*b*> & *z* = {*cons*(<*n*,*b*>, *f*)}}
apply (*simp add*: *succ-fun-eq*)
apply (*blast dest*: *transM*)
done

lemma (**in** *M-basic*) *funspace-succ*:
[|*M*(*n*); *M*(*B*); *M*(*n*−>*B*) |] ==> *M*(*succ*(*n*) −> *B*)

apply (*insert funspace-succ-replacement* [*of n*], *simp*)
apply (*force simp add*: *succ-fun-eq2 univalent-def*)
done

M contains all finite function spaces. Needed to prove the absoluteness of transitive closure. See the definition of *rtrancl-alt* in in *WF-absolute.thy*.

lemma (**in** *M-basic*) *finite-funspace-closed* [*intro*,*simp*]:
[|*n*∈*nat*; *M*(*B*)|] ==> *M*(*n*−>*B*)
apply (*induct-tac n*, *simp*)
apply (*simp add*: *funspace-succ nat-into-M*)
done

1.8 Relativization and Absoluteness for Boolean Operators

definition
is-bool-of-o :: [*i*=>*o*, *o*, *i*] => *o* **where**
is-bool-of-o(*M*,*P*,*z*) == (*P* & *number1*(*M*,*z*)) | (~*P* & *empty*(*M*,*z*))

definition
is-not :: [*i*=>*o*, *i*, *i*] => *o* **where**
is-not(*M*,*a*,*z*) == (*number1*(*M*,*a*) & *empty*(*M*,*z*)) |
(~*number1*(*M*,*a*) & *number1*(*M*,*z*))

definition
is-and :: [*i*=>*o*, *i*, *i*, *i*] => *o* **where**
is-and(*M*,*a*,*b*,*z*) == (*number1*(*M*,*a*) & *z*=*b*) |
(~*number1*(*M*,*a*) & *empty*(*M*,*z*))

definition
is-or :: [*i*=>*o*, *i*, *i*, *i*] => *o* **where**
is-or(*M*,*a*,*b*,*z*) == (*number1*(*M*,*a*) & *number1*(*M*,*z*)) |
(~*number1*(*M*,*a*) & *z*=*b*)

lemma (**in** *M-trivial*) *bool-of-o-abs* [*simp*]:
M(*z*) ==> *is-bool-of-o*(*M*,*P*,*z*) ⟷ *z* = *bool-of-o*(*P*)
by (*simp add*: *is-bool-of-o-def bool-of-o-def*)

lemma (**in** *M-trivial*) *not-abs* [*simp*]:
[| *M*(*a*); *M*(*z*)|] ==> *is-not*(*M*,*a*,*z*) ⟷ *z* = *not*(*a*)
by (*simp add*: *Bool.not-def cond-def is-not-def*)

lemma (**in** *M-trivial*) *and-abs* [*simp*]:
[| *M*(*a*); *M*(*b*); *M*(*z*)|] ==> *is-and*(*M*,*a*,*b*,*z*) ⟷ *z* = *a and b*
by (*simp add*: *Bool.and-def cond-def is-and-def*)

lemma (**in** *M-trivial*) *or-abs* [*simp*]:
[| *M*(*a*); *M*(*b*); *M*(*z*)|] ==> *is-or*(*M*,*a*,*b*,*z*) ⟷ *z* = *a or b*
by (*simp add*: *Bool.or-def cond-def is-or-def*)

lemma (**in** *M-trivial*) *bool-of-o-closed* [*intro*,*simp*]:
  M(*bool-of-o*(*P*))
by (*simp add*: *bool-of-o-def*)

lemma (**in** *M-trivial*) *and-closed* [*intro*,*simp*]:
  [| *M*(*p*); *M*(*q*) |] ==> *M*(*p and q*)
by (*simp add*: *and-def cond-def*)

lemma (**in** *M-trivial*) *or-closed* [*intro*,*simp*]:
  [| *M*(*p*); *M*(*q*) |] ==> *M*(*p or q*)
by (*simp add*: *or-def cond-def*)

lemma (**in** *M-trivial*) *not-closed* [*intro*,*simp*]:
  M(*p*) ==> *M*(*not*(*p*))
by (*simp add*: *Bool.not-def cond-def*)

1.9 Relativization and Absoluteness for List Operators

definition
 is-Nil :: [*i*=>*o*, *i*] => *o* **where**
  — because [] ≡ *Inl*(*0*)
  is-Nil(*M*,*xs*) == ∃ *zero*[*M*]. *empty*(*M*,*zero*) & *is-Inl*(*M*,*zero*,*xs*)

definition
 is-Cons :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
  — because *Cons*(*a*, *l*) ≡ *Inr*(⟨*a*, *l*⟩)
  is-Cons(*M*,*a*,*l*,*Z*) == ∃ *p*[*M*]. *pair*(*M*,*a*,*l*,*p*) & *is-Inr*(*M*,*p*,*Z*)

lemma (**in** *M-trivial*) *Nil-in-M* [*intro*,*simp*]: *M*(*Nil*)
by (*simp add*: *Nil-def*)

lemma (**in** *M-trivial*) *Nil-abs* [*simp*]: *M*(*Z*) ==> *is-Nil*(*M*,*Z*) ⟷ (*Z* = *Nil*)
by (*simp add*: *is-Nil-def Nil-def*)

lemma (**in** *M-trivial*) *Cons-in-M-iff* [*iff*]: *M*(*Cons*(*a*,*l*)) ⟷ *M*(*a*) & *M*(*l*)
by (*simp add*: *Cons-def*)

lemma (**in** *M-trivial*) *Cons-abs* [*simp*]:
  [|*M*(*a*); *M*(*l*); *M*(*Z*)|] ==> *is-Cons*(*M*,*a*,*l*,*Z*) ⟷ (*Z* = *Cons*(*a*,*l*))
by (*simp add*: *is-Cons-def Cons-def*)

definition
 quasilist :: *i* => *o* **where**
  quasilist(*xs*) == *xs*=*Nil* | (∃ *x l*. *xs* = *Cons*(*x*,*l*))

definition
is-quasilist :: [*i*=>*o*,*i*] => *o* **where**
is-quasilist(*M*,*z*) == *is-Nil*(*M*,*z*) | (∃ *x*[*M*]. ∃ *l*[*M*]. *is-Cons*(*M*,*x*,*l*,*z*))

definition
list-case' :: [*i*, [*i*,*i*]=>*i*, *i*] => *i* **where**
— A version of *list-case* that's always defined.
list-case'(*a*,*b*,*xs*) ==
if quasilist(*xs*) *then list-case*(*a*,*b*,*xs*) *else 0*

definition
is-list-case :: [*i*=>*o*, *i*, [*i*,*i*,*i*]=>*o*, *i*, *i*] => *o* **where**
— Returns 0 for non-lists
is-list-case(*M*, *a*, *is-b*, *xs*, *z*) ==
(*is-Nil*(*M*,*xs*) ⟶ *z*=*a*) &
(∀ *x*[*M*]. ∀ *l*[*M*]. *is-Cons*(*M*,*x*,*l*,*xs*) ⟶ *is-b*(*x*,*l*,*z*)) &
(*is-quasilist*(*M*,*xs*) | *empty*(*M*,*z*))

definition
hd' :: *i* => *i* **where**
— A version of *hd* that's always defined.
hd'(*xs*) == *if quasilist*(*xs*) *then hd*(*xs*) *else 0*

definition
tl' :: *i* => *i* **where**
— A version of *tl* that's always defined.
tl'(*xs*) == *if quasilist*(*xs*) *then tl*(*xs*) *else 0*

definition
is-hd :: [*i*=>*o*,*i*,*i*] => *o* **where**
— *hd*([]) = *0* no constraints if not a list. Avoiding implication prevents the simplifier's looping.
is-hd(*M*,*xs*,*H*) ==
(*is-Nil*(*M*,*xs*) ⟶ *empty*(*M*,*H*)) &
(∀ *x*[*M*]. ∀ *l*[*M*]. ~ *is-Cons*(*M*,*x*,*l*,*xs*) | *H*=*x*) &
(*is-quasilist*(*M*,*xs*) | *empty*(*M*,*H*))

definition
is-tl :: [*i*=>*o*,*i*,*i*] => *o* **where**
— *tl*([]) = []; see comments about *is-hd*
is-tl(*M*,*xs*,*T*) ==
(*is-Nil*(*M*,*xs*) ⟶ *T*=*xs*) &
(∀ *x*[*M*]. ∀ *l*[*M*]. ~ *is-Cons*(*M*,*x*,*l*,*xs*) | *T*=*l*) &
(*is-quasilist*(*M*,*xs*) | *empty*(*M*,*T*))

1.9.1 *quasilist*: For Case-Splitting with *list-case'*

lemma [*iff*]: *quasilist*(*Nil*)
by (*simp add*: *quasilist-def*)

lemma [*iff*]: *quasilist*(*Cons*(*x*,*l*))
by (*simp add*: *quasilist-def*)

lemma *list-imp-quasilist*: *l* ∈ *list*(*A*) ==> *quasilist*(*l*)
by (*erule list.cases*, *simp-all*)

1.9.2 *list-case′*, the Modified Version of *list-case*

lemma *list-case′-Nil* [*simp*]: *list-case′*(*a*,*b*,*Nil*) = *a*
by (*simp add*: *list-case′-def quasilist-def*)

lemma *list-case′-Cons* [*simp*]: *list-case′*(*a*,*b*,*Cons*(*x*,*l*)) = *b*(*x*,*l*)
by (*simp add*: *list-case′-def quasilist-def*)

lemma *non-list-case*: ~ *quasilist*(*x*) ==> *list-case′*(*a*,*b*,*x*) = *0*
by (*simp add*: *quasilist-def list-case′-def*)

lemma *list-case′-eq-list-case* [*simp*]:
    xs ∈ *list*(*A*) ==>*list-case′*(*a*,*b*,*xs*) = *list-case*(*a*,*b*,*xs*)
by (*erule list.cases*, *simp-all*)

lemma (**in** *M-basic*) *list-case′-closed* [*intro*,*simp*]:
  [|*M*(*k*); *M*(*a*); ∀ *x*[*M*]. ∀ *y*[*M*]. *M*(*b*(*x*,*y*))|] ==> *M*(*list-case′*(*a*,*b*,*k*))
apply (*case-tac quasilist*(*k*))
 apply (*simp add*: *quasilist-def*, *force*)
apply (*simp add*: *non-list-case*)
done

lemma (**in** *M-trivial*) *quasilist-abs* [*simp*]:
    M(*z*) ==> *is-quasilist*(*M*,*z*) ⟷ *quasilist*(*z*)
by (*auto simp add*: *is-quasilist-def quasilist-def*)

lemma (**in** *M-trivial*) *list-case-abs* [*simp*]:
    [| *relation2*(*M*,*is-b*,*b*); *M*(*k*); *M*(*z*) |]
     ==> *is-list-case*(*M*,*a*,*is-b*,*k*,*z*) ⟷ *z* = *list-case′*(*a*,*b*,*k*)
apply (*case-tac quasilist*(*k*))
 prefer *2*
 apply (*simp add*: *is-list-case-def non-list-case*)
 apply (*force simp add*: *quasilist-def*)
apply (*simp add*: *quasilist-def is-list-case-def*)
apply (*elim disjE exE*)
 apply (*simp-all add*: *relation2-def*)
done

1.9.3 The Modified Operators *hd′* and *tl′*

lemma (**in** *M-trivial*) *is-hd-Nil*: *is-hd*(*M*,[],*Z*) ⟷ *empty*(*M*,*Z*)
by (*simp add*: *is-hd-def*)

lemma (**in** *M-trivial*) *is-hd-Cons*:
  $[|M(a); M(l)|] ==> is\text{-}hd(M,Cons(a,l),Z) \longleftrightarrow Z = a$
by (*force simp add*: *is-hd-def*)

lemma (**in** *M-trivial*) *hd-abs* [*simp*]:
  $[|M(x); M(y)|] ==> is\text{-}hd(M,x,y) \longleftrightarrow y = hd'(x)$
apply (*simp add*: *hd′-def*)
apply (*intro impI conjI*)
 prefer *2* **apply** (*force simp add*: *is-hd-def*)
apply (*simp add*: *quasilist-def is-hd-def*)
apply (*elim disjE exE*, *auto*)
done

lemma (**in** *M-trivial*) *is-tl-Nil*: $is\text{-}tl(M,[],Z) \longleftrightarrow Z = []$
by (*simp add*: *is-tl-def*)

lemma (**in** *M-trivial*) *is-tl-Cons*:
  $[|M(a); M(l)|] ==> is\text{-}tl(M,Cons(a,l),Z) \longleftrightarrow Z = l$
by (*force simp add*: *is-tl-def*)

lemma (**in** *M-trivial*) *tl-abs* [*simp*]:
  $[|M(x); M(y)|] ==> is\text{-}tl(M,x,y) \longleftrightarrow y = tl'(x)$
apply (*simp add*: *tl′-def*)
apply (*intro impI conjI*)
 prefer *2* **apply** (*force simp add*: *is-tl-def*)
apply (*simp add*: *quasilist-def is-tl-def*)
apply (*elim disjE exE*, *auto*)
done

lemma (**in** *M-trivial*) *relation1-tl*: $relation1(M, is\text{-}tl(M), tl')$
by (*simp add*: *relation1-def*)

lemma *hd′-Nil*: $hd'([]) = 0$
by (*simp add*: *hd′-def*)

lemma *hd′-Cons*: $hd'(Cons(a,l)) = a$
by (*simp add*: *hd′-def*)

lemma *tl′-Nil*: $tl'([]) = []$
by (*simp add*: *tl′-def*)

lemma *tl′-Cons*: $tl'(Cons(a,l)) = l$
by (*simp add*: *tl′-def*)

lemma *iterates-tl-Nil*: $n \in nat ==> tl'\hat{~}n\ ([]) = []$
apply (*induct-tac n*)
apply (*simp-all add*: *tl′-Nil*)
done

lemma (**in** *M-basic*) *tl′-closed*: $M(x) ==> M(tl'(x))$
apply (*simp add*: *tl′-def*)
apply (*force simp add*: *quasilist-def*)
done

end

2 First-Order Formulas and the Definition of the Class L

theory *Formula* **imports** *ZF* **begin**

2.1 Internalized formulas of FOL

De Bruijn representation. Unbound variables get their denotations from an environment.

```
consts  formula :: i
datatype
  formula = Member (x ∈ nat, y ∈ nat)
          | Equal  (x ∈ nat, y ∈ nat)
          | Nand (p ∈ formula, q ∈ formula)
          | Forall (p ∈ formula)

declare formula.intros [TC]

definition
  Neg :: i=>i where
  Neg(p) == Nand(p,p)

definition
  And :: [i,i]=>i where
  And(p,q) == Neg(Nand(p,q))

definition
  Or :: [i,i]=>i where
  Or(p,q) == Nand(Neg(p),Neg(q))

definition
  Implies :: [i,i]=>i where
  Implies(p,q) == Nand(p,Neg(q))

definition
  Iff :: [i,i]=>i where
  Iff(p,q) == And(Implies(p,q), Implies(q,p))

definition
  Exists :: i=>i where
```

Exists(*p*) == *Neg*(*Forall*(*Neg*(*p*)))

lemma *Neg-type* [*TC*]: *p* ∈ *formula* ==> *Neg*(*p*) ∈ *formula*
by (*simp add*: *Neg-def*)

lemma *And-type* [*TC*]: [| *p* ∈ *formula*; *q* ∈ *formula* |] ==> *And*(*p*,*q*) ∈ *formula*
by (*simp add*: *And-def*)

lemma *Or-type* [*TC*]: [| *p* ∈ *formula*; *q* ∈ *formula* |] ==> *Or*(*p*,*q*) ∈ *formula*
by (*simp add*: *Or-def*)

lemma *Implies-type* [*TC*]:
[| *p* ∈ *formula*; *q* ∈ *formula* |] ==> *Implies*(*p*,*q*) ∈ *formula*
by (*simp add*: *Implies-def*)

lemma *Iff-type* [*TC*]:
[| *p* ∈ *formula*; *q* ∈ *formula* |] ==> *Iff*(*p*,*q*) ∈ *formula*
by (*simp add*: *Iff-def*)

lemma *Exists-type* [*TC*]: *p* ∈ *formula* ==> *Exists*(*p*) ∈ *formula*
by (*simp add*: *Exists-def*)

consts *satisfies* :: [*i*,*i*]=>*i*
primrec
satisfies(*A*,*Member*(*x*,*y*)) =
(λ*env* ∈ *list*(*A*). *bool-of-o* (*nth*(*x*,*env*) ∈ *nth*(*y*,*env*)))

satisfies(*A*,*Equal*(*x*,*y*)) =
(λ*env* ∈ *list*(*A*). *bool-of-o* (*nth*(*x*,*env*) = *nth*(*y*,*env*)))

satisfies(*A*,*Nand*(*p*,*q*)) =
(λ*env* ∈ *list*(*A*). *not* ((*satisfies*(*A*,*p*)‘*env*) *and* (*satisfies*(*A*,*q*)‘*env*)))

satisfies(*A*,*Forall*(*p*)) =
(λ*env* ∈ *list*(*A*). *bool-of-o* (∀ *x*∈*A*. *satisfies*(*A*,*p*) ‘ (*Cons*(*x*,*env*)) = *1*))

lemma *p* ∈ *formula* ==> *satisfies*(*A*,*p*) ∈ *list*(*A*) −> *bool*
by (*induct set*: *formula*) *simp-all*

abbreviation
sats :: [*i*,*i*,*i*] => *o* **where**
sats(*A*,*p*,*env*) == *satisfies*(*A*,*p*)‘*env* = *1*

lemma [*simp*]:
env ∈ *list*(*A*)
==> *sats*(*A*, *Member*(*x*,*y*), *env*) ⟷ *nth*(*x*,*env*) ∈ *nth*(*y*,*env*)
by *simp*

lemma [*simp*]:
  env ∈ *list*(*A*)
  ==> *sats*(*A*, *Equal*(*x*,*y*), *env*) ⟷ *nth*(*x*,*env*) = *nth*(*y*,*env*)
by *simp*

lemma *sats-Nand-iff* [*simp*]:
  env ∈ *list*(*A*)
  ==> (*sats*(*A*, *Nand*(*p*,*q*), *env*)) ⟷ ~ (*sats*(*A*,*p*,*env*) & *sats*(*A*,*q*,*env*))
by (*simp add*: *Bool.and-def Bool.not-def cond-def*)

lemma *sats-Forall-iff* [*simp*]:
  env ∈ *list*(*A*)
  ==> *sats*(*A*, *Forall*(*p*), *env*) ⟷ (∀ *x*∈*A*. *sats*(*A*, *p*, *Cons*(*x*,*env*)))
by *simp*

declare *satisfies.simps* [*simp del*]

2.2 Dividing line between primitive and derived connectives

lemma *sats-Neg-iff* [*simp*]:
  env ∈ *list*(*A*)
  ==> *sats*(*A*, *Neg*(*p*), *env*) ⟷ ~ *sats*(*A*,*p*,*env*)
by (*simp add*: *Neg-def*)

lemma *sats-And-iff* [*simp*]:
  env ∈ *list*(*A*)
  ==> (*sats*(*A*, *And*(*p*,*q*), *env*)) ⟷ *sats*(*A*,*p*,*env*) & *sats*(*A*,*q*,*env*)
by (*simp add*: *And-def*)

lemma *sats-Or-iff* [*simp*]:
  env ∈ *list*(*A*)
  ==> (*sats*(*A*, *Or*(*p*,*q*), *env*)) ⟷ *sats*(*A*,*p*,*env*) | *sats*(*A*,*q*,*env*)
by (*simp add*: *Or-def*)

lemma *sats-Implies-iff* [*simp*]:
  env ∈ *list*(*A*)
  ==> (*sats*(*A*, *Implies*(*p*,*q*), *env*)) ⟷ (*sats*(*A*,*p*,*env*) ⟶ *sats*(*A*,*q*,*env*))
by (*simp add*: *Implies-def*, *blast*)

lemma *sats-Iff-iff* [*simp*]:
  env ∈ *list*(*A*)
  ==> (*sats*(*A*, *Iff*(*p*,*q*), *env*)) ⟷ (*sats*(*A*,*p*,*env*) ⟷ *sats*(*A*,*q*,*env*))
by (*simp add*: *Iff-def*, *blast*)

lemma *sats-Exists-iff* [*simp*]:
  env ∈ *list*(*A*)
  ==> *sats*(*A*, *Exists*(*p*), *env*) ⟷ (∃ *x*∈*A*. *sats*(*A*, *p*, *Cons*(*x*,*env*)))
by (*simp add*: *Exists-def*)

2.2.1 Derived rules to help build up formulas

lemma *mem-iff-sats*:
  [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *env* ∈ *list*(*A*)|]
  ==> (*x*∈*y*) ⟷ *sats*(*A*, *Member*(*i*,*j*), *env*)
by (*simp add*: *satisfies.simps*)

lemma *equal-iff-sats*:
  [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *env* ∈ *list*(*A*)|]
  ==> (*x*=*y*) ⟷ *sats*(*A*, *Equal*(*i*,*j*), *env*)
by (*simp add*: *satisfies.simps*)

lemma *not-iff-sats*:
  [| *P* ⟷ *sats*(*A*,*p*,*env*); *env* ∈ *list*(*A*)|]
  ==> (~*P*) ⟷ *sats*(*A*, *Neg*(*p*), *env*)
by *simp*

lemma *conj-iff-sats*:
  [| *P* ⟷ *sats*(*A*,*p*,*env*); *Q* ⟷ *sats*(*A*,*q*,*env*); *env* ∈ *list*(*A*)|]
  ==> (*P* & *Q*) ⟷ *sats*(*A*, *And*(*p*,*q*), *env*)
by (*simp add*: *sats-And-iff*)

lemma *disj-iff-sats*:
  [| *P* ⟷ *sats*(*A*,*p*,*env*); *Q* ⟷ *sats*(*A*,*q*,*env*); *env* ∈ *list*(*A*)|]
  ==> (*P* | *Q*) ⟷ *sats*(*A*, *Or*(*p*,*q*), *env*)
by (*simp add*: *sats-Or-iff*)

lemma *iff-iff-sats*:
  [| *P* ⟷ *sats*(*A*,*p*,*env*); *Q* ⟷ *sats*(*A*,*q*,*env*); *env* ∈ *list*(*A*)|]
  ==> (*P* ⟷ *Q*) ⟷ *sats*(*A*, *Iff*(*p*,*q*), *env*)
by (*simp add*: *sats-Forall-iff*)

lemma *imp-iff-sats*:
  [| *P* ⟷ *sats*(*A*,*p*,*env*); *Q* ⟷ *sats*(*A*,*q*,*env*); *env* ∈ *list*(*A*)|]
  ==> (*P* ⟶ *Q*) ⟷ *sats*(*A*, *Implies*(*p*,*q*), *env*)
by (*simp add*: *sats-Forall-iff*)

lemma *ball-iff-sats*:
  [| !!*x*. *x*∈*A* ==> *P*(*x*) ⟷ *sats*(*A*, *p*, *Cons*(*x*, *env*)); *env* ∈ *list*(*A*)|]
  ==> (∀*x*∈*A*. *P*(*x*)) ⟷ *sats*(*A*, *Forall*(*p*), *env*)
by (*simp add*: *sats-Forall-iff*)

lemma *bex-iff-sats*:
  [| !!*x*. *x*∈*A* ==> *P*(*x*) ⟷ *sats*(*A*, *p*, *Cons*(*x*, *env*)); *env* ∈ *list*(*A*)|]
  ==> (∃*x*∈*A*. *P*(*x*)) ⟷ *sats*(*A*, *Exists*(*p*), *env*)
by (*simp add*: *sats-Exists-iff*)

lemmas *FOL-iff-sats* =
  mem-iff-sats equal-iff-sats not-iff-sats conj-iff-sats
  disj-iff-sats imp-iff-sats iff-iff-sats imp-iff-sats ball-iff-sats

bex-iff-sats

2.3 Arity of a Formula: Maximum Free de Bruijn Index

consts *arity* :: *i=>i*
primrec
arity(*Member*(*x*,*y*)) = *succ*(*x*) ∪ *succ*(*y*)

arity(*Equal*(*x*,*y*)) = *succ*(*x*) ∪ *succ*(*y*)

arity(*Nand*(*p*,*q*)) = *arity*(*p*) ∪ *arity*(*q*)

arity(*Forall*(*p*)) = *Arith.pred*(*arity*(*p*))

lemma *arity-type* [*TC*]: *p* ∈ *formula* ==> *arity*(*p*) ∈ *nat*
by (*induct-tac p*, *simp-all*)

lemma *arity-Neg* [*simp*]: *arity*(*Neg*(*p*)) = *arity*(*p*)
by (*simp add*: *Neg-def*)

lemma *arity-And* [*simp*]: *arity*(*And*(*p*,*q*)) = *arity*(*p*) ∪ *arity*(*q*)
by (*simp add*: *And-def*)

lemma *arity-Or* [*simp*]: *arity*(*Or*(*p*,*q*)) = *arity*(*p*) ∪ *arity*(*q*)
by (*simp add*: *Or-def*)

lemma *arity-Implies* [*simp*]: *arity*(*Implies*(*p*,*q*)) = *arity*(*p*) ∪ *arity*(*q*)
by (*simp add*: *Implies-def*)

lemma *arity-Iff* [*simp*]: *arity*(*Iff*(*p*,*q*)) = *arity*(*p*) ∪ *arity*(*q*)
by (*simp add*: *Iff-def*, *blast*)

lemma *arity-Exists* [*simp*]: *arity*(*Exists*(*p*)) = *Arith.pred*(*arity*(*p*))
by (*simp add*: *Exists-def*)

lemma *arity-sats-iff* [*rule-format*]:
[| *p* ∈ *formula*; *extra* ∈ *list*(*A*) |]
==> ∀ *env* ∈ *list*(*A*).
arity(*p*) ≤ *length*(*env*) ⟶
sats(*A*, *p*, *env* @ *extra*) ⟷ *sats*(*A*, *p*, *env*)
apply (*induct-tac p*)
apply (*simp-all add*: *Arith.pred-def nth-append Un-least-lt-iff nat-imp-quasinat*
split: *split-nat-case*, *auto*)
done

lemma *arity-sats1-iff*:
[| *arity*(*p*) ≤ *succ*(*length*(*env*)); *p* ∈ *formula*; *x* ∈ *A*; *env* ∈ *list*(*A*);

extra ∈ *list*(*A*) |]
==> *sats*(*A*, *p*, *Cons*(*x*, *env* @ *extra*)) ⟷ *sats*(*A*, *p*, *Cons*(*x*, *env*))
apply (*insert arity-sats-iff* [*of p extra A Cons*(*x*,*env*)])
apply *simp*
done

2.4 Renaming Some de Bruijn Variables

definition
incr-var :: [*i*,*i*]=>*i* **where**
incr-var(*x*,*nq*) == *if x*<*nq then x else succ*(*x*)

lemma *incr-var-lt*: *x*<*nq* ==> *incr-var*(*x*,*nq*) = *x*
by (*simp add*: *incr-var-def*)

lemma *incr-var-le*: *nq*≤*x* ==> *incr-var*(*x*,*nq*) = *succ*(*x*)
apply (*simp add*: *incr-var-def*)
apply (*blast dest*: *lt-trans1*)
done

consts *incr-bv* :: *i*=>*i*
primrec
incr-bv(*Member*(*x*,*y*)) =
(λ*nq* ∈ *nat*. *Member* (*incr-var*(*x*,*nq*), *incr-var*(*y*,*nq*)))

incr-bv(*Equal*(*x*,*y*)) =
(λ*nq* ∈ *nat*. *Equal* (*incr-var*(*x*,*nq*), *incr-var*(*y*,*nq*)))

incr-bv(*Nand*(*p*,*q*)) =
(λ*nq* ∈ *nat*. *Nand* (*incr-bv*(*p*)‘*nq*, *incr-bv*(*q*)‘*nq*))

incr-bv(*Forall*(*p*)) =
(λ*nq* ∈ *nat*. *Forall* (*incr-bv*(*p*) ‘ *succ*(*nq*)))

lemma [*TC*]: *x* ∈ *nat* ==> *incr-var*(*x*,*nq*) ∈ *nat*
by (*simp add*: *incr-var-def*)

lemma *incr-bv-type* [*TC*]: *p* ∈ *formula* ==> *incr-bv*(*p*) ∈ *nat* −> *formula*
by (*induct-tac p*, *simp-all*)

Obviously, *DPow* is closed under complements and finite intersections and unions. Needs an inductive lemma to allow two lists of parameters to be combined.

lemma *sats-incr-bv-iff* [*rule-format*]:
[| *p* ∈ *formula*; *env* ∈ *list*(*A*); *x* ∈ *A* |]
==> ∀ *bvs* ∈ *list*(*A*).
sats(*A*, *incr-bv*(*p*) ‘ *length*(*bvs*), *bvs* @ *Cons*(*x*,*env*)) ⟷
sats(*A*, *p*, *bvs*@*env*)

apply (*induct-tac p*)
apply (*simp-all add*: *incr-var-def nth-append succ-lt-iff length-type*)
apply (*auto simp add*: *diff-succ not-lt-iff-le*)
done

lemma *incr-var-lemma*:
[| $x \in nat$; $y \in nat$; $nq \leq x$ |]
==> $succ(x) \cup incr\text{-}var(y,nq) = succ(x \cup y)$
apply (*simp add*: *incr-var-def Ord-Un-if*, *auto*)
apply (*blast intro*: *leI*)
apply (*simp add*: *not-lt-iff-le*)
apply (*blast intro*: *le-anti-sym*)
apply (*blast dest*: *lt-trans2*)
done

lemma *incr-And-lemma*:
$y < x$ ==> $y \cup succ(x) = succ(x \cup y)$
apply (*simp add*: *Ord-Un-if lt-Ord lt-Ord2 succ-lt-iff*)
apply (*blast dest*: *lt-asym*)
done

lemma *arity-incr-bv-lemma* [*rule-format*]:
$p \in formula$
==> $\forall n \in nat.$ *arity* (*incr-bv*(*p*) ' *n*) =
(*if* $n < arity(p)$ *then* *succ*(*arity*(*p*)) *else* *arity*(*p*))
apply (*induct-tac p*)
apply (*simp-all add*: *imp-disj not-lt-iff-le Un-least-lt-iff lt-Un-iff le-Un-iff*
succ-Un-distrib [*symmetric*] *incr-var-lt incr-var-le*
Un-commute incr-var-lemma Arith.pred-def nat-imp-quasinat
split: *split-nat-case*)

the Forall case reduces to linear arithmetic

prefer *2*
apply *clarify*
apply (*blast dest*: *lt-trans1*)

left with the And case

apply *safe*
apply (*blast intro*: *incr-And-lemma lt-trans1*)
apply (*subst incr-And-lemma*)
apply (*blast intro*: *lt-trans1*)
apply (*simp add*: *Un-commute*)
done

2.5 Renaming all but the First de Bruijn Variable

definition

```
incr-bv1 :: i => i where
  incr-bv1(p) == incr-bv(p)`1


lemma incr-bv1-type [TC]: p ∈ formula ==> incr-bv1(p) ∈ formula
by (simp add: incr-bv1-def)


lemma sats-incr-bv1-iff:
  [| p ∈ formula; env ∈ list(A); x ∈ A; y ∈ A |]
   ==> sats(A, incr-bv1(p), Cons(x, Cons(y, env))) ⟷
       sats(A, p, Cons(x,env))
apply (insert sats-incr-bv-iff [of p env A y Cons(x,Nil)])
apply (simp add: incr-bv1-def)
done

lemma formula-add-params1 [rule-format]:
  [| p ∈ formula; n ∈ nat; x ∈ A |]
   ==> ∀ bvs ∈ list(A). ∀ env ∈ list(A).
          length(bvs) = n ⟶
          sats(A, iterates(incr-bv1, n, p), Cons(x, bvs@env)) ⟷
          sats(A, p, Cons(x,env))
apply (induct-tac n, simp, clarify)
apply (erule list.cases)
apply (simp-all add: sats-incr-bv1-iff)
done


lemma arity-incr-bv1-eq:
  p ∈ formula
   ==> arity(incr-bv1(p)) =
        (if 1 < arity(p) then succ(arity(p)) else arity(p))
apply (insert arity-incr-bv-lemma [of p 1])
apply (simp add: incr-bv1-def)
done

lemma arity-iterates-incr-bv1-eq:
  [| p ∈ formula; n ∈ nat |]
   ==> arity(incr-bv1^n(p)) =
         (if 1 < arity(p) then n #+ arity(p) else arity(p))
apply (induct-tac n)
apply (simp-all add: arity-incr-bv1-eq)
apply (simp add: not-lt-iff-le)
apply (blast intro: le-trans add-le-self2 arity-type)
done
```

2.6 Definable Powerset

The definable powerset operation: Kunen's definition VI 1.1, page 165.

definition
  DPow :: *i* => *i* **where**
  $DPow(A) == \{X \in Pow(A).$
              $\exists\, env \in list(A).\ \exists\, p \in formula.$
                $arity(p) \leq succ(length(env))\ \&$
                $X = \{x{\in}A.\ sats(A,\ p,\ Cons(x,env))\}\}$

lemma *DPowI*:
  $[|env \in list(A);\ \ p \in formula;\ \ arity(p) \leq succ(length(env))|]$
  $==> \{x{\in}A.\ sats(A,\ p,\ Cons(x,env))\} \in DPow(A)$
by (*simp add*: *DPow-def*, *blast*)

With this rule we can specify p later.

lemma *DPowI2* [*rule-format*]:
  $[|\forall\, x{\in}A.\ P(x) \longleftrightarrow sats(A,\ p,\ Cons(x,env));$
    $env \in list(A);\ \ p \in formula;\ \ arity(p) \leq succ(length(env))|]$
  $==> \{x{\in}A.\ P(x)\} \in DPow(A)$
by (*simp add*: *DPow-def*, *blast*)

lemma *DPowD*:
  $X \in DPow(A)$
  $==> X \subseteq A\ \&$
    $(\exists\, env \in list(A).$
     $\exists\, p \in formula.\ arity(p) \leq succ(length(env))\ \&$
            $X = \{x{\in}A.\ sats(A,\ p,\ Cons(x,env))\})$
by (*simp add*: *DPow-def*)

lemmas *DPow-imp-subset* = *DPowD* [*THEN conjunct1*]

lemma $[|\ p \in formula;\ env \in list(A);\ arity(p) \leq succ(length(env))\ |]$
    $==> \{x{\in}A.\ sats(A,\ p,\ Cons(x,env))\} \in DPow(A)$
by (*blast intro*: *DPowI*)

lemma *DPow-subset-Pow*: $DPow(A) \subseteq Pow(A)$
by (*simp add*: *DPow-def*, *blast*)

lemma *empty-in-DPow*: $0 \in DPow(A)$
apply (*simp add*: *DPow-def*)
apply (*rule-tac x=Nil* **in** *bexI*)
 apply (*rule-tac x=Neg*(*Equal*(*0*,*0*)) **in** *bexI*)
  apply (*auto simp add*: *Un-least-lt-iff*)
done

lemma *Compl-in-DPow*: $X \in DPow(A) ==> (A-X) \in DPow(A)$
apply (*simp add*: *DPow-def*, *clarify*, *auto*)
apply (*rule bexI*)
 apply (*rule-tac x=Neg*(*p*) **in** *bexI*)
  apply *auto*

done

lemma *Int-in-DPow*: [| $X \in DPow(A)$; $Y \in DPow(A)$ |] ==> $X \cap Y \in DPow(A)$
apply (*simp add*: *DPow-def*, *auto*)
apply (*rename-tac envp p envq q*)
apply (*rule-tac x=envp@envq* **in** *bexI*)
 apply (*rule-tac x=And(p, iterates(incr-bv1,length(envp),q))* **in** *bexI*)
  apply *typecheck*
apply (*rule conjI*)

 apply (*simp add*: *arity-iterates-incr-bv1-eq length-app Un-least-lt-iff*)
 apply (*force intro*: *add-le-self le-trans*)
apply (*simp add*: *arity-sats1-iff formula-add-params1*, *blast*)
done

lemma *Un-in-DPow*: [| $X \in DPow(A)$; $Y \in DPow(A)$ |] ==> $X \cup Y \in DPow(A)$
apply (*subgoal-tac* $X \cup Y = A - ((A-X) \cap (A-Y))$)
apply (*simp add*: *Int-in-DPow Compl-in-DPow*)
apply (*simp add*: *DPow-def*, *blast*)
done

lemma *singleton-in-DPow*: $a \in A$ ==> $\{a\} \in DPow(A)$
apply (*simp add*: *DPow-def*)
apply (*rule-tac x=Cons(a,Nil)* **in** *bexI*)
 apply (*rule-tac x=Equal(0,1)* **in** *bexI*)
  apply *typecheck*
apply (*force simp add*: *succ-Un-distrib* [*symmetric*])
done

lemma *cons-in-DPow*: [| $a \in A$; $X \in DPow(A)$ |] ==> $cons(a,X) \in DPow(A)$
apply (*rule cons-eq* [*THEN subst*])
apply (*blast intro*: *singleton-in-DPow Un-in-DPow*)
done

lemma *Fin-into-DPow*: $X \in Fin(A)$ ==> $X \in DPow(A)$
apply (*erule Fin.induct*)
 apply (*rule empty-in-DPow*)
apply (*blast intro*: *cons-in-DPow*)
done

DPow is not monotonic. For example, let A be some non-constructible set of natural numbers, and let B be *nat*. Then $A \subseteq B$ and obviously $A \in DPow(A)$ but $A \notin DPow(B)$.

lemma *Finite-Pow-subset-Pow*: $Finite(A)$ ==> $Pow(A) \subseteq DPow(A)$
by (*blast intro*: *Fin-into-DPow Finite-into-Fin Fin-subset*)

lemma *Finite-DPow-eq-Pow*: $Finite(A)$ ==> $DPow(A) = Pow(A)$
apply (*rule equalityI*)

apply (*rule DPow-subset-Pow*)
apply (*erule Finite-Pow-subset-Pow*)
done

2.7 Internalized Formulas for the Ordinals

The *sats* theorems below differ from the usual form in that they include an element of absoluteness. That is, they relate internalized formulas to real concepts such as the subset relation, rather than to the relativized concepts defined in theory *Relative*. This lets us prove the theorem as *Ords-in-DPow* without first having to instantiate the locale *M-trivial*. Note that the present theory does not even take *Relative* as a parent.

2.7.1 The subset relation

definition
subset-fm :: [*i*,*i*]=>*i* **where**
subset-fm(*x*,*y*) == *Forall*(*Implies*(*Member*(*0*,*succ*(*x*)), *Member*(*0*,*succ*(*y*))))

lemma *subset-type* [*TC*]: [| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *subset-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *subset-fm-def*)

lemma *arity-subset-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *arity*(*subset-fm*(*x*,*y*)) = *succ*(*x*) ∪ *succ*(*y*)
by (*simp add*: *subset-fm-def succ-Un-distrib* [*symmetric*])

lemma *sats-subset-fm* [*simp*]:
[|*x* < *length*(*env*); *y* ∈ *nat*; *env* ∈ *list*(*A*); *Transset*(*A*)|]
==> *sats*(*A*, *subset-fm*(*x*,*y*), *env*) ⟷ *nth*(*x*,*env*) ⊆ *nth*(*y*,*env*)
apply (*frule lt-length-in-nat*, *assumption*)
apply (*simp add*: *subset-fm-def Transset-def*)
apply (*blast intro*: *nth-type*)
done

2.7.2 Transitive sets

definition
transset-fm :: *i*=>*i* **where**
transset-fm(*x*) == *Forall*(*Implies*(*Member*(*0*,*succ*(*x*)), *subset-fm*(*0*,*succ*(*x*))))

lemma *transset-type* [*TC*]: *x* ∈ *nat* ==> *transset-fm*(*x*) ∈ *formula*
by (*simp add*: *transset-fm-def*)

lemma *arity-transset-fm* [*simp*]:
x ∈ *nat* ==> *arity*(*transset-fm*(*x*)) = *succ*(*x*)
by (*simp add*: *transset-fm-def succ-Un-distrib* [*symmetric*])

lemma *sats-transset-fm* [*simp*]:

[|*x* < *length*(*env*); *env* ∈ *list*(*A*); *Transset*(*A*)|]
==> *sats*(*A*, *transset-fm*(*x*), *env*) ⟷ *Transset*(*nth*(*x*,*env*))
apply (*frule lt-nat-in-nat*, *erule length-type*)
apply (*simp add*: *transset-fm-def Transset-def*)
apply (*blast intro*: *nth-type*)
done

2.7.3 Ordinals

definition
ordinal-fm :: *i*=>*i* **where**
ordinal-fm(*x*) ==
And(*transset-fm*(*x*), *Forall*(*Implies*(*Member*(*0*,*succ*(*x*)), *transset-fm*(*0*))))

lemma *ordinal-type* [*TC*]: *x* ∈ *nat* ==> *ordinal-fm*(*x*) ∈ *formula*
by (*simp add*: *ordinal-fm-def*)

lemma *arity-ordinal-fm* [*simp*]:
x ∈ *nat* ==> *arity*(*ordinal-fm*(*x*)) = *succ*(*x*)
by (*simp add*: *ordinal-fm-def succ-Un-distrib* [*symmetric*])

lemma *sats-ordinal-fm*:
[|*x* < *length*(*env*); *env* ∈ *list*(*A*); *Transset*(*A*)|]
==> *sats*(*A*, *ordinal-fm*(*x*), *env*) ⟷ *Ord*(*nth*(*x*,*env*))
apply (*frule lt-nat-in-nat*, *erule length-type*)
apply (*simp add*: *ordinal-fm-def Ord-def Transset-def*)
apply (*blast intro*: *nth-type*)
done

The subset consisting of the ordinals is definable. Essential lemma for *Ord-in-Lset*. This result is the objective of the present subsection.

theorem *Ords-in-DPow*: *Transset*(*A*) ==> {*x* ∈ *A*. *Ord*(*x*)} ∈ *DPow*(*A*)
apply (*simp add*: *DPow-def Collect-subset*)
apply (*rule-tac x*=*Nil* **in** *bexI*)
apply (*rule-tac x*=*ordinal-fm*(*0*) **in** *bexI*)
apply (*simp-all add*: *sats-ordinal-fm*)
done

2.8 Constant Lset: Levels of the Constructible Universe

definition
Lset :: *i*=>*i* **where**
Lset(*i*) == *transrec*(*i*, %*x f*. ⋃ *y*∈*x*. *DPow*(*f*‘*y*))

definition
L :: *i*=>*o* **where** — Kunen's definition VI 1.5, page 167
L(*x*) == ∃ *i*. *Ord*(*i*) & *x* ∈ *Lset*(*i*)

NOT SUITABLE FOR REWRITING – RECURSIVE!

lemma *Lset*: *Lset*(*i*) = ($\bigcup$*j*∈*i*. *DPow*(*Lset*(*j*)))
by (*subst Lset-def* [*THEN def-transrec*], *simp*)

lemma *LsetI*: [|*y*∈*x*; *A* ∈ *DPow*(*Lset*(*y*))|] ==> *A* ∈ *Lset*(*x*)
by (*subst Lset*, *blast*)

lemma *LsetD*: *A* ∈ *Lset*(*x*) ==> ∃ *y*∈*x*. *A* ∈ *DPow*(*Lset*(*y*))
apply (*insert Lset* [*of x*])
apply (*blast intro*: *elim*: *equalityE*)
done

2.8.1 Transitivity

lemma *elem-subset-in-DPow*: [|*X* ∈ *A*; *X* ⊆ *A*|] ==> *X* ∈ *DPow*(*A*)
apply (*simp add*: *Transset-def DPow-def*)
apply (*rule-tac x*=[*X*] **in** *bexI*)
 apply (*rule-tac x*=*Member*(*0*,*1*) **in** *bexI*)
  apply (*auto simp add*: *Un-least-lt-iff*)
done

lemma *Transset-subset-DPow*: *Transset*(*A*) ==> *A* ⊆ *DPow*(*A*)
apply *clarify*
apply (*simp add*: *Transset-def*)
apply (*blast intro*: *elem-subset-in-DPow*)
done

lemma *Transset-DPow*: *Transset*(*A*) ==> *Transset*(*DPow*(*A*))
apply (*simp add*: *Transset-def*)
apply (*blast intro*: *elem-subset-in-DPow dest*: *DPowD*)
done

Kunen's VI 1.6 (a)

lemma *Transset-Lset*: *Transset*(*Lset*(*i*))
apply (*rule-tac a*=*i* **in** *eps-induct*)
apply (*subst Lset*)
apply (*blast intro*!: *Transset-Union-family Transset-Un Transset-DPow*)
done

lemma *mem-Lset-imp-subset-Lset*: *a* ∈ *Lset*(*i*) ==> *a* ⊆ *Lset*(*i*)
apply (*insert Transset-Lset*)
apply (*simp add*: *Transset-def*)
done

2.8.2 Monotonicity

Kunen's VI 1.6 (b)

lemma *Lset-mono* [*rule-format*]:
  ∀ *j*. *i*<=*j* ⟶ *Lset*(*i*) ⊆ *Lset*(*j*)
proof (*induct i rule*: *eps-induct*, *intro allI impI*)

fix $x\ j$
assume $\forall y{\in}x.\ \forall j.\ y \subseteq j \longrightarrow Lset(y) \subseteq Lset(j)$
and $x \subseteq j$
thus $Lset(x) \subseteq Lset(j)$
by (*force simp add*: *Lset* [*of x*] *Lset* [*of j*])
qed

This version lets us remove the premise $Ord(i)$ sometimes.

lemma *Lset-mono-mem* [*rule-format*]:
$\forall j.\ i \in j \longrightarrow Lset(i) \subseteq Lset(j)$
proof (*induct i rule*: *eps-induct*, *intro allI impI*)
fix $x\ j$
assume $\forall y{\in}x.\ \forall j.\ y \in j \longrightarrow Lset(y) \subseteq Lset(j)$
and $x \in j$
thus $Lset(x) \subseteq Lset(j)$
by (*force simp add*: *Lset* [*of j*]
intro!: *bexI intro*: *elem-subset-in-DPow dest*: *LsetD DPowD*)
qed

Useful with Reflection to bump up the ordinal

lemma *subset-Lset-ltD*: $[|A \subseteq Lset(i);\ i < j|] ==> A \subseteq Lset(j)$
by (*blast dest*: *ltD* [*THEN Lset-mono-mem*])

2.8.3 0, successor and limit equations for Lset

lemma *Lset-0* [*simp*]: $Lset(0) = 0$
by (*subst Lset*, *blast*)

lemma *Lset-succ-subset1*: $DPow(Lset(i)) \subseteq Lset(succ(i))$
by (*subst Lset*, *rule succI1* [*THEN RepFunI*, *THEN Union-upper*])

lemma *Lset-succ-subset2*: $Lset(succ(i)) \subseteq DPow(Lset(i))$
apply (*subst Lset*, *rule UN-least*)
apply (*erule succE*)
apply *blast*
apply *clarify*
apply (*rule elem-subset-in-DPow*)
apply (*subst Lset*)
apply *blast*
apply (*blast intro*: *dest*: *DPowD Lset-mono-mem*)
done

lemma *Lset-succ*: $Lset(succ(i)) = DPow(Lset(i))$
by (*intro equalityI Lset-succ-subset1 Lset-succ-subset2*)

lemma *Lset-Union* [*simp*]: $Lset(\bigcup(X)) = (\bigcup y{\in}X.\ Lset(y))$
apply (*subst Lset*)
apply (*rule equalityI*)

first inclusion

apply (*rule UN-least*)
apply (*erule UnionE*)
apply (*rule subset-trans*)
apply (*erule-tac* [*2*] *UN-upper*, *subst Lset*, *erule UN-upper*)

opposite inclusion

apply (*rule UN-least*)
apply (*subst Lset*, *blast*)
done

2.8.4 Lset applied to Limit ordinals

lemma *Limit-Lset-eq*:
    Limit(*i*) ==> *Lset*(*i*) = (⋃ *y*∈*i*. *Lset*(*y*))
by (*simp add*: *Lset-Union* [*symmetric*] *Limit-Union-eq*)

lemma *lt-LsetI*: [| *a* ∈ *Lset*(*j*); *j*<*i* |] ==> *a* ∈ *Lset*(*i*)
by (*blast dest*: *Lset-mono* [*OF le-imp-subset* [*OF leI*]])

lemma *Limit-LsetE*:
    [| *a* ∈ *Lset*(*i*); ~*R* ==> *Limit*(*i*);
        !!*x*. [| *x*<*i*; *a* ∈ *Lset*(*x*) |] ==> *R*
    |] ==> *R*
apply (*rule classical*)
apply (*rule Limit-Lset-eq* [*THEN equalityD1*, *THEN subsetD*, *THEN UN-E*])
prefer *2* **apply** *assumption*
apply *blast*
apply (*blast intro*: *ltI Limit-is-Ord*)
done

2.8.5 Basic closure properties

lemma *zero-in-Lset*: *y* ∈ *x* ==> *0* ∈ *Lset*(*x*)
by (*subst Lset*, *blast intro*: *empty-in-DPow*)

lemma *notin-Lset*: *x* ∉ *Lset*(*x*)
apply (*rule-tac a*=*x* **in** *eps-induct*)
apply (*subst Lset*)
apply (*blast dest*: *DPowD*)
done

2.9 Constructible Ordinals: Kunen's VI 1.9 (b)

lemma *Ords-of-Lset-eq*: *Ord*(*i*) ==> {*x*∈*Lset*(*i*). *Ord*(*x*)} = *i*
apply (*erule trans-induct3*)
apply (*simp-all add*: *Lset-succ Limit-Lset-eq Limit-Union-eq*)

The successor case remains.

apply (*rule equalityI*)

First inclusion

apply *clarify*
apply (*erule Ord-linear-lt*, *assumption*)
apply (*blast dest*: *DPow-imp-subset ltD notE* [*OF notin-Lset*])
apply *blast*
apply (*blast dest*: *ltD*)

Opposite inclusion, $succ(x) \subseteq DPow(Lset(x)) \cap ON$

apply *auto*

Key case:

apply (*erule subst*, *rule Ords-in-DPow* [*OF Transset-Lset*])
apply (*blast intro*: *elem-subset-in-DPow dest*: *OrdmemD elim*: *equalityE*)
apply (*blast intro*: *Ord-in-Ord*)
done

lemma *Ord-subset-Lset*: *Ord*(i) ==> $i \subseteq$ *Lset*(i)
by (*subst Ords-of-Lset-eq* [*symmetric*], *assumption*, *fast*)

lemma *Ord-in-Lset*: *Ord*(i) ==> $i \in$ *Lset*(*succ*(i))
apply (*simp add*: *Lset-succ*)
apply (*subst Ords-of-Lset-eq* [*symmetric*], *assumption*,
rule Ords-in-DPow [*OF Transset-Lset*])
done

lemma *Ord-in-L*: *Ord*(i) ==> *L*(i)
by (*simp add*: *L-def*, *blast intro*: *Ord-in-Lset*)

2.9.1 Unions

lemma *Union-in-Lset*:
$X \in$ *Lset*(i) ==> $\bigcup(X) \in$ *Lset*(*succ*(i))
apply (*insert Transset-Lset*)
apply (*rule LsetI* [*OF succI1*])
apply (*simp add*: *Transset-def DPow-def*)
apply (*intro conjI*, *blast*)

Now to create the formula $\exists y.\ y \in X \land x \in y$

apply (*rule-tac x*=*Cons*(*X*,*Nil*) **in** *bexI*)
apply (*rule-tac x*=*Exists*(*And*(*Member*(*0*,*2*), *Member*(*1*,*0*))) **in** *bexI*)
apply *typecheck*
apply (*simp add*: *succ-Un-distrib* [*symmetric*], *blast*)
done

theorem *Union-in-L*: *L*(X) ==> *L*($\bigcup(X)$)
by (*simp add*: *L-def*, *blast dest*: *Union-in-Lset*)

2.9.2 Finite sets and ordered pairs

lemma *singleton-in-Lset*: *a* ∈ *Lset*(*i*) ==> {*a*} ∈ *Lset*(*succ*(*i*))
by (*simp add*: *Lset-succ singleton-in-DPow*)

lemma *doubleton-in-Lset*:
  [| *a* ∈ *Lset*(*i*); *b* ∈ *Lset*(*i*) |] ==> {*a*,*b*} ∈ *Lset*(*succ*(*i*))
by (*simp add*: *Lset-succ empty-in-DPow cons-in-DPow*)

lemma *Pair-in-Lset*:
  [| *a* ∈ *Lset*(*i*); *b* ∈ *Lset*(*i*); *Ord*(*i*) |] ==> <*a*,*b*> ∈ *Lset*(*succ*(*succ*(*i*)))
apply (*unfold Pair-def*)
apply (*blast intro*: *doubleton-in-Lset*)
done

lemmas *Lset-UnI1* = *Un-upper1* [*THEN Lset-mono* [*THEN subsetD*]]
lemmas *Lset-UnI2* = *Un-upper2* [*THEN Lset-mono* [*THEN subsetD*]]

Hard work is finding a single *j* ∈ *i* such that {*a*, *b*} ⊆ *Lset*(*j*)

lemma *doubleton-in-LLimit*:
  [| *a* ∈ *Lset*(*i*); *b* ∈ *Lset*(*i*); *Limit*(*i*) |] ==> {*a*,*b*} ∈ *Lset*(*i*)
apply (*erule Limit-LsetE*, *assumption*)
apply (*erule Limit-LsetE*, *assumption*)
apply (*blast intro*: *lt-LsetI* [*OF doubleton-in-Lset*]
    Lset-UnI1 Lset-UnI2 Limit-has-succ Un-least-lt)
done

theorem *doubleton-in-L*: [| *L*(*a*); *L*(*b*) |] ==> *L*({*a*, *b*})
apply (*simp add*: *L-def*, *clarify*)
apply (*drule Ord2-imp-greater-Limit*, *assumption*)
apply (*blast intro*: *lt-LsetI doubleton-in-LLimit Limit-is-Ord*)
done

lemma *Pair-in-LLimit*:
  [| *a* ∈ *Lset*(*i*); *b* ∈ *Lset*(*i*); *Limit*(*i*) |] ==> <*a*,*b*> ∈ *Lset*(*i*)

Infer that a, b occur at ordinals x,xa ¡ i.

apply (*erule Limit-LsetE*, *assumption*)
apply (*erule Limit-LsetE*, *assumption*)

Infer that *succ*(*succ*(*x* ∪ *xa*)) < *i*

apply (*blast intro*: *lt-Ord lt-LsetI* [*OF Pair-in-Lset*]
    Lset-UnI1 Lset-UnI2 Limit-has-succ Un-least-lt)
done

The rank function for the constructible universe

definition
 lrank :: *i*=>*i* **where** — Kunen's definition VI 1.7
 lrank(*x*) == *μ i*. *x* ∈ *Lset*(*succ*(*i*))

lemma *L-I*: $[|x \in Lset(i); Ord(i)|] ==> L(x)$
by (*simp add*: *L-def*, *blast*)

lemma *L-D*: $L(x) ==> \exists\, i.\ Ord(i)\ \&\ x \in Lset(i)$
by (*simp add*: *L-def*)

lemma *Ord-lrank* [*simp*]: $Ord(lrank(a))$
by (*simp add*: *lrank-def*)

lemma *Lset-lrank-lt* [*rule-format*]: $Ord(i) ==> x \in Lset(i) \longrightarrow lrank(x) < i$
apply (*erule trans-induct3*)
  apply *simp*
 apply (*simp only*: *lrank-def*)
 apply (*blast intro*: *Least-le*)
apply (*simp-all add*: *Limit-Lset-eq*)
apply (*blast intro*: *ltI Limit-is-Ord lt-trans*)
done

Kunen's VI 1.8. The proof is much harder than the text would suggest. For a start, it needs the previous lemma, which is proved by induction.

lemma *Lset-iff-lrank-lt*: $Ord(i) ==> x \in Lset(i) \longleftrightarrow L(x)\ \&\ lrank(x) < i$
apply (*simp add*: *L-def*, *auto*)
 apply (*blast intro*: *Lset-lrank-lt*)
 apply (*unfold lrank-def*)
apply (*drule succI1* [*THEN Lset-mono-mem*, *THEN subsetD*])
apply (*drule-tac* $P = \lambda i.\ x \in Lset(succ(i))$ **in** *LeastI*, *assumption*)
apply (*blast intro*!: *le-imp-subset Lset-mono* [*THEN subsetD*])
done

lemma *Lset-succ-lrank-iff* [*simp*]: $x \in Lset(succ(lrank(x))) \longleftrightarrow L(x)$
by (*simp add*: *Lset-iff-lrank-lt*)

Kunen's VI 1.9 (a)

lemma *lrank-of-Ord*: $Ord(i) ==> lrank(i) = i$
apply (*unfold lrank-def*)
apply (*rule Least-equality*)
  apply (*erule Ord-in-Lset*)
 apply *assumption*
apply (*insert notin-Lset* [*of i*])
apply (*blast intro*!: *le-imp-subset Lset-mono* [*THEN subsetD*])
done

This is lrank(lrank(a)) = lrank(a)

declare *Ord-lrank* [*THEN lrank-of-Ord*, *simp*]

Kunen's VI 1.10

lemma *Lset-in-Lset-succ*: $Lset(i) \in Lset(succ(i))$

apply (*simp add*: *Lset-succ DPow-def*)
apply (*rule-tac x=Nil* **in** *bexI*)
apply (*rule-tac x=Equal(0,0)* **in** *bexI*)
apply *auto*
done

lemma *lrank-Lset*: *Ord*(*i*) ==> *lrank*(*Lset*(*i*)) = *i*
apply (*unfold lrank-def*)
apply (*rule Least-equality*)
apply (*rule Lset-in-Lset-succ*)
apply *assumption*
apply *clarify*
apply (*subgoal-tac Lset*(*succ*(*ia*)) ⊆ *Lset*(*i*))
apply (*blast dest*: *mem-irrefl*)
apply (*blast intro*!: *le-imp-subset Lset-mono*)
done

Kunen's VI 1.11

lemma *Lset-subset-Vset*: *Ord*(*i*) ==> *Lset*(*i*) ⊆ *Vset*(*i*)
apply (*erule trans-induct*)
apply (*subst Lset*)
apply (*subst Vset*)
apply (*rule UN-mono* [*OF subset-refl*])
apply (*rule subset-trans* [*OF DPow-subset-Pow*])
apply (*rule Pow-mono*, *blast*)
done

Kunen's VI 1.12

lemma *Lset-subset-Vset'*: *i* ∈ *nat* ==> *Lset*(*i*) = *Vset*(*i*)
apply (*erule nat-induct*)
apply (*simp add*: *Vfrom-0*)
apply (*simp add*: *Lset-succ Vset-succ Finite-Vset Finite-DPow-eq-Pow*)
done

Every set of constructible sets is included in some *Lset*

lemma *subset-Lset*:
(∀ *x*∈*A*. *L*(*x*)) ==> ∃ *i*. *Ord*(*i*) & *A* ⊆ *Lset*(*i*)
by (*rule-tac x* = ⋃ *x*∈*A*. *succ*(*lrank*(*x*)) **in** *exI*, *force*)

lemma *subset-LsetE*:
[|∀ *x*∈*A*. *L*(*x*);
!!*i*. [|*Ord*(*i*); *A* ⊆ *Lset*(*i*)|] ==> *P*|]
==> *P*
by (*blast dest*: *subset-Lset*)

2.9.3 For L to satisfy the Powerset axiom

lemma *LPow-env-typing*:
[| *y* ∈ *Lset*(*i*); *Ord*(*i*); *y* ⊆ *X* |]

==> ∃ *z* ∈ *Pow*(*X*). *y* ∈ *Lset*(*succ*(*lrank*(*z*)))
by (*auto intro*: *L-I iff*: *Lset-succ-lrank-iff*)

lemma *LPow-in-Lset*:
[|*X* ∈ *Lset*(*i*); *Ord*(*i*)|] ==> ∃ *j*. *Ord*(*j*) & {*y* ∈ *Pow*(*X*). *L*(*y*)} ∈ *Lset*(*j*)
apply (*rule-tac x=succ*(⋃ *y* ∈ *Pow*(*X*). *succ*(*lrank*(*y*))) **in** *exI*)
apply *simp*
apply (*rule LsetI* [*OF succI1*])
apply (*simp add*: *DPow-def*)
apply (*intro conjI*, *clarify*)
apply (*rule-tac a=x* **in** *UN-I*, *simp+*)

Now to create the formula $y \subseteq X$

apply (*rule-tac x=Cons*(*X*,*Nil*) **in** *bexI*)
apply (*rule-tac x=subset-fm*(*0*,*1*) **in** *bexI*)
apply *typecheck*
apply (*rule conjI*)
apply (*simp add*: *succ-Un-distrib* [*symmetric*])
apply (*rule equality-iffI*)
apply (*simp add*: *Transset-UN* [*OF Transset-Lset*] *LPow-env-typing*)
apply (*auto intro*: *L-I iff*: *Lset-succ-lrank-iff*)
done

theorem *LPow-in-L*: *L*(*X*) ==> *L*({*y* ∈ *Pow*(*X*). *L*(*y*)})
by (*blast intro*: *L-I dest*: *L-D LPow-in-Lset*)

2.10 Eliminating *arity* from the Definition of *Lset*

lemma *nth-zero-eq-0*: *n* ∈ *nat* ==> *nth*(*n*,[*0*]) = *0*
by (*induct-tac n*, *auto*)

lemma *sats-app-0-iff* [*rule-format*]:
[| *p* ∈ *formula*; *0* ∈ *A* |]
==> ∀ *env* ∈ *list*(*A*). *sats*(*A*,*p*, *env*@[*0*]) ⟷ *sats*(*A*,*p*,*env*)
apply (*induct-tac p*)
apply (*simp-all del*: *app-Cons add*: *app-Cons* [*symmetric*]
add: *nth-zero-eq-0 nth-append not-lt-iff-le nth-eq-0*)
done

lemma *sats-app-zeroes-iff*:
[| *p* ∈ *formula*; *0* ∈ *A*; *env* ∈ *list*(*A*); *n* ∈ *nat* |]
==> *sats*(*A*,*p*,*env* @ *repeat*(*0*,*n*)) ⟷ *sats*(*A*,*p*,*env*)
apply (*induct-tac n*, *simp*)
apply (*simp del*: *repeat.simps*
add: *repeat-succ-app sats-app-0-iff app-assoc* [*symmetric*])
done

lemma *exists-bigger-env*:
[| *p* ∈ *formula*; *0* ∈ *A*; *env* ∈ *list*(*A*) |]
==> ∃ *env'* ∈ *list*(*A*). *arity*(*p*) ≤ *succ*(*length*(*env'*)) &

(∀ a∈A. sats(A,p,Cons(a,env')) ⟷ sats(A,p,Cons(a,env)))
apply (*rule-tac x=env @ repeat(0,arity(p))* **in** *bexI*)
apply (*simp del*: *app-Cons add*: *app-Cons* [*symmetric*]
    add: *length-repeat sats-app-zeroes-iff*, *typecheck*)
done

A simpler version of *DPow*: no arity check!

definition
  DPow' :: *i* => *i* **where**
  DPow'(A) == {*X* ∈ *Pow(A)*.
      ∃ *env* ∈ *list(A)*. ∃ *p* ∈ *formula*.
        X = {*x*∈*A*. *sats(A, p, Cons(x,env))*}}

lemma *DPow-subset-DPow'*: *DPow(A)* ⊆ *DPow'(A)*
by (*simp add*: *DPow-def DPow'-def*, *blast*)

lemma *DPow'-0*: *DPow'(0)* = {*0*}
by (*auto simp add*: *DPow'-def*)

lemma *DPow'-subset-DPow*: *0* ∈ *A* ==> *DPow'(A)* ⊆ *DPow(A)*
apply (*auto simp add*: *DPow'-def DPow-def*)
apply (*frule exists-bigger-env*, *assumption+*, *force*)
done

lemma *DPow-eq-DPow'*: *Transset(A)* ==> *DPow(A)* = *DPow'(A)*
apply (*drule Transset-0-disj*)
apply (*erule disjE*)
 apply (*simp add*: *DPow'-0 Finite-DPow-eq-Pow*)
apply (*rule equalityI*)
 apply (*rule DPow-subset-DPow'*)
apply (*erule DPow'-subset-DPow*)
done

And thus we can relativize *Lset* without bothering with *arity* and *length*

lemma *Lset-eq-transrec-DPow'*: *Lset(i)* = *transrec(i, %x f. ⋃ y∈x. DPow'(f`y))*
apply (*rule-tac a=i* **in** *eps-induct*)
apply (*subst Lset*)
apply (*subst transrec*)
apply (*simp only*: *DPow-eq-DPow'* [*OF Transset-Lset*], *simp*)
done

With this rule we can specify *p* later and don't worry about arities at all!

lemma *DPow-LsetI* [*rule-format*]:
  [|∀ *x*∈*Lset(i)*. *P(x)* ⟷ *sats(Lset(i), p, Cons(x,env))*;
    env ∈ *list(Lset(i))*; *p* ∈ *formula*|]
  ==> {*x*∈*Lset(i)*. *P(x)*} ∈ *DPow(Lset(i))*
by (*simp add*: *DPow-eq-DPow'* [*OF Transset-Lset*] *DPow'-def*, *blast*)

end

```
theory Forcing-Data
  imports
    Forcing-Notions
    Relative
    ~~/src/ZF/Constructible/Formula

begin

lemma lam-codomain: ∀ n∈N. (λx∈N. b(x))`n ∈ B ⟹ (λx∈N. b(x)) : N→B
  apply (rule fun-weaken-type)
   apply (subgoal-tac (λx∈N. b(x)) : N → {b(x).x∈N}, assumption)
   apply (auto simp add:lam-funtype)
  done

lemma Transset-M :
  Transset(M) ⟹ y∈x ⟹ x ∈ M ⟹ y ∈ M
  by (simp add: Transset-def,auto)

definition
  infinity-ax :: (i ⇒ o) ⇒ o where
  infinity-ax(M) ==
     (∃ I[M]. (∃ z[M]. empty(M,z) ∧ z∈I) ∧ (∀ y[M]. y∈I ⟶ (∃ sy[M]. succes-
sor(M,y,sy) ∧ sy∈I)))

locale M-ZF =
  fixes M
  assumes
         upair-ax:          upair-ax(##M)
      and Union-ax:          Union-ax(##M)
      and power-ax:          power-ax(##M)
      and extensionality:    extensionality(##M)
      and foundation-ax:     foundation-ax(##M)
      and infinity-ax:       infinity-ax(##M)
      and separation-ax:        ⟦ φ ∈ formula ; arity(φ)=1 ∨ arity(φ)=2 ⟧ ⟹
                          (∀ a∈M. separation(##M,λx. sats(M,φ,[x,a])))
      and replacement-ax:       ⟦ φ ∈ formula ; arity(φ)=2 ∨ arity(φ)=succ(2) ⟧
⟹
                     (∀ a∈M. strong-replacement(##M,λx y. sats(M,φ,[x,y,a])))

locale forcing-data = forcing-notion + M-ZF +
  fixes enum
  assumes M-countable:      enum∈bij(nat,M)
      and P-in-M:           P ∈ M
      and leq-in-M:         leq ∈ M
      and trans-M:          Transset(M)

begin
definition
  M-generic :: i⇒o where
```

M-generic(*G*) == *filter*(*G*) ∧ (∀ *D*∈*M*. *D*⊆*P* ∧ *dense*(*D*)⟶*D*∩*G*≠*0*)

lemma *G-nonempty*: *M-generic*(*G*) ⟹ *G*≠*0*
proof −
  have *P*⊆*P* **..**
  assume
    M-generic(*G*)
  with *P-in-M P-dense* ‹*P*⊆*P*› **show**
    G ≠ *0*
    unfolding *M-generic-def* **by** *auto*
qed

lemma *one-in-G* :
  assumes *M-generic*(*G*)
  shows *one* ∈ *G*
proof −
  from *assms* **have** *G*⊆*P*
    unfolding *M-generic-def* **and** *filter-def* **by** *simp*
  from ‹*M-generic*(*G*)› **have** *increasing*(*G*)
    unfolding *M-generic-def* **and** *filter-def* **by** *simp*
  with ‹*G*⊆*P*› **and** ‹*M-generic*(*G*)›
  show *?thesis*
    using *G-nonempty* **and** *one-in-P* **and** *one-max*
    unfolding *increasing-def* **by** *blast*
qed

declare *iff-trans* [*trans*]

lemma *generic-filter-existence*:
  p∈*P* ⟹ ∃ *G*. *p*∈*G* ∧ *M-generic*(*G*)
proof −
  assume
      Eq1: *p*∈*P*
  let
      ?D=λ*n*∈*nat*. (*if* (*enum*‘*n*⊆*P* ∧ *dense*(*enum*‘*n*)) *then enum*‘*n else P*)
  have
      Eq2: ∀ *n*∈*nat*. *?D*‘*n* ∈ *Pow*(*P*)
    by *auto*
  then have
      Eq3: *?D*:*nat*→*Pow*(*P*)
    by (*rule lam-codomain*)
  have
      Eq4: ∀ *n*∈*nat*. *dense*(*?D*‘*n*)
  proof
    show
        dense(*?D*‘*n*)
    if *Eq5*: *n*∈*nat*   **for** *n*
    proof −
      have

```
          dense(?D`n)
          ⟷ dense(if enum ` n ⊆ P ∧ dense(enum ` n) then enum ` n else P)
      using Eq5 by simp
    also have
          ... ⟷ (¬(enum ` n ⊆ P ∧ dense(enum ` n)) ⟶ dense(P))
      using split-if by simp
    finally show ?thesis
      using P-dense and Eq5 by auto
  qed
qed
from Eq3 and Eq4 interpret
      cg: countable-generic P leq one ?D
  by (unfold-locales, auto)
from cg.rasiowa-sikorski and Eq1 obtain G where
      Eq6: p∈G ∧ filter(G) ∧ (∀ n∈nat.(?D`n)∩G≠0)
  unfolding cg.D-generic-def by blast
then have
      Eq7: (∀ D∈M. D⊆P ∧ dense(D)⟶D∩G≠0)
proof (intro ballI impI)
  show
        D ∩ G ≠ 0
  if  Eq8: D∈M and
      Eq9: D ⊆ P ∧ dense(D)  for D
  proof −
    from M-countable and  bij-is-surj have
          ∀ y∈M. ∃ x∈nat. enum`x= y
      unfolding surj-def  by (simp)
    with Eq8 obtain n where
      Eq10: n∈nat ∧ enum`n = D
      by auto
    with Eq9 and if-P have
      Eq11: ?D`n = D
      by (simp)
    with Eq6 and Eq10 show
          D∩G≠0
      by auto
  qed
  with Eq6 have
      Eq12: ∃ G. filter(G) ∧ (∀ D∈M. D⊆P ∧ dense(D)⟶D∩G≠0)
    by auto
qed
with Eq6 show ?thesis
  unfolding M-generic-def by auto
qed


end

end
```

3 Relativized Wellorderings

theory *Wellorderings* **imports** *Relative* **begin**

We define functions analogous to *ordermap ordertype* but without using recursion. Instead, there is a direct appeal to Replacement. This will be the basis for a version relativized to some class *M*. The main result is Theorem I 7.6 in Kunen, page 17.

3.1 Wellorderings

definition
irreflexive :: [*i*=>*o*,*i*,*i*]=>*o* **where**
irreflexive(*M*,*A*,*r*) == ∀ *x*[*M*]. *x*∈*A* ⟶ <*x*,*x*> ∉ *r*

definition
transitive-rel :: [*i*=>*o*,*i*,*i*]=>*o* **where**
transitive-rel(*M*,*A*,*r*) ==
∀ *x*[*M*]. *x*∈*A* ⟶ (∀ *y*[*M*]. *y*∈*A* ⟶ (∀ *z*[*M*]. *z*∈*A* ⟶
<*x*,*y*>∈*r* ⟶ <*y*,*z*>∈*r* ⟶ <*x*,*z*>∈*r*))

definition
linear-rel :: [*i*=>*o*,*i*,*i*]=>*o* **where**
linear-rel(*M*,*A*,*r*) ==
∀ *x*[*M*]. *x*∈*A* ⟶ (∀ *y*[*M*]. *y*∈*A* ⟶ <*x*,*y*>∈*r* | *x*=*y* | <*y*,*x*>∈*r*)

definition
wellfounded :: [*i*=>*o*,*i*]=>*o* **where**
— EVERY non-empty set has an *r*-minimal element
wellfounded(*M*,*r*) ==
∀ *x*[*M*]. *x*≠*0* ⟶ (∃ *y*[*M*]. *y*∈*x* & ~(∃ *z*[*M*]. *z*∈*x* & <*z*,*y*> ∈ *r*))

definition
wellfounded-on :: [*i*=>*o*,*i*,*i*]=>*o* **where**
— every non-empty SUBSET OF *A* has an *r*-minimal element
wellfounded-on(*M*,*A*,*r*) ==
∀ *x*[*M*]. *x*≠*0* ⟶ *x*⊆*A* ⟶ (∃ *y*[*M*]. *y*∈*x* & ~(∃ *z*[*M*]. *z*∈*x* & <*z*,*y*> ∈ *r*))

definition
wellordered :: [*i*=>*o*,*i*,*i*]=>*o* **where**
— linear and wellfounded on *A*
wellordered(*M*,*A*,*r*) ==
transitive-rel(*M*,*A*,*r*) & *linear-rel*(*M*,*A*,*r*) & *wellfounded-on*(*M*,*A*,*r*)

3.1.1 Trivial absoluteness proofs

lemma (**in** *M-basic*) *irreflexive-abs* [*simp*]:
M(*A*) ==> *irreflexive*(*M*,*A*,*r*) ⟷ *irrefl*(*A*,*r*)
by (*simp add*: *irreflexive-def irrefl-def*)

lemma (**in** *M-basic*) *transitive-rel-abs* [*simp*]:
M(*A*) ==> *transitive-rel*(*M*,*A*,*r*) ⟷ *trans*[*A*](*r*)
by (*simp add*: *transitive-rel-def trans-on-def*)

lemma (**in** *M-basic*) *linear-rel-abs* [*simp*]:
M(*A*) ==> *linear-rel*(*M*,*A*,*r*) ⟷ *linear*(*A*,*r*)
by (*simp add*: *linear-rel-def linear-def*)

lemma (**in** *M-basic*) *wellordered-is-trans-on*:
[| *wellordered*(*M*,*A*,*r*); *M*(*A*) |] ==> *trans*[*A*](*r*)
by (*auto simp add*: *wellordered-def*)

lemma (**in** *M-basic*) *wellordered-is-linear*:
[| *wellordered*(*M*,*A*,*r*); *M*(*A*) |] ==> *linear*(*A*,*r*)
by (*auto simp add*: *wellordered-def*)

lemma (**in** *M-basic*) *wellordered-is-wellfounded-on*:
[| *wellordered*(*M*,*A*,*r*); *M*(*A*) |] ==> *wellfounded-on*(*M*,*A*,*r*)
by (*auto simp add*: *wellordered-def*)

lemma (**in** *M-basic*) *wellfounded-imp-wellfounded-on*:
[| *wellfounded*(*M*,*r*); *M*(*A*) |] ==> *wellfounded-on*(*M*,*A*,*r*)
by (*auto simp add*: *wellfounded-def wellfounded-on-def*)

lemma (**in** *M-basic*) *wellfounded-on-subset-A*:
[| *wellfounded-on*(*M*,*A*,*r*); *B*<=*A* |] ==> *wellfounded-on*(*M*,*B*,*r*)
by (*simp add*: *wellfounded-on-def*, *blast*)

3.1.2 Well-founded relations

lemma (**in** *M-basic*) *wellfounded-on-iff-wellfounded*:
wellfounded-on(*M*,*A*,*r*) ⟷ *wellfounded*(*M*, *r* ∩ *A*∗*A*)
apply (*simp add*: *wellfounded-on-def wellfounded-def*, *safe*)
apply *force*
apply (*drule-tac x*=*x* **in** *rspec*, *assumption*, *blast*)
done

lemma (**in** *M-basic*) *wellfounded-on-imp-wellfounded*:
[|*wellfounded-on*(*M*,*A*,*r*); *r* ⊆ *A*∗*A*|] ==> *wellfounded*(*M*,*r*)
by (*simp add*: *wellfounded-on-iff-wellfounded subset-Int-iff*)

lemma (**in** *M-basic*) *wellfounded-on-field-imp-wellfounded*:
wellfounded-on(*M*, *field*(*r*), *r*) ==> *wellfounded*(*M*,*r*)
by (*simp add*: *wellfounded-def wellfounded-on-iff-wellfounded*, *fast*)

lemma (**in** *M-basic*) *wellfounded-iff-wellfounded-on-field*:
M(*r*) ==> *wellfounded*(*M*,*r*) ⟷ *wellfounded-on*(*M*, *field*(*r*), *r*)
by (*blast intro*: *wellfounded-imp-wellfounded-on*
wellfounded-on-field-imp-wellfounded)

lemma (**in** *M-basic*) *wellfounded-induct*:
  [| *wellfounded*(*M*,*r*); *M*(*a*); *M*(*r*); *separation*(*M*, λ*x*. ~*P*(*x*));
    ∀ *x*. *M*(*x*) & (∀ *y*. <*y*,*x*> ∈ *r* ⟶ *P*(*y*)) ⟶ *P*(*x*) |]
  ==> *P*(*a*)
apply (*simp* (*no-asm-use*) *add*: *wellfounded-def*)
apply (*drule-tac x*={*z* ∈ *domain*(*r*). ~*P*(*z*)} **in** *rspec*)
apply (*blast dest*: *transM*)+
done

lemma (**in** *M-basic*) *wellfounded-on-induct*:
  [| *a*∈*A*; *wellfounded-on*(*M*,*A*,*r*); *M*(*A*);
    separation(*M*, λ*x*. *x*∈*A* ⟶ ~*P*(*x*));
    ∀ *x*∈*A*. *M*(*x*) & (∀ *y*∈*A*. <*y*,*x*> ∈ *r* ⟶ *P*(*y*)) ⟶ *P*(*x*) |]
  ==> *P*(*a*)
apply (*simp* (*no-asm-use*) *add*: *wellfounded-on-def*)
apply (*drule-tac x*={*z*∈*A*. *z*∈*A* ⟶ ~*P*(*z*)} **in** *rspec*)
apply (*blast intro*: *transM*)+
done

3.1.3 Kunen's lemma IV 3.14, page 123

lemma (**in** *M-basic*) *linear-imp-relativized*:
  linear(*A*,*r*) ==> *linear-rel*(*M*,*A*,*r*)
by (*simp add*: *linear-def linear-rel-def*)

lemma (**in** *M-basic*) *trans-on-imp-relativized*:
  trans[*A*](*r*) ==> *transitive-rel*(*M*,*A*,*r*)
by (*unfold transitive-rel-def trans-on-def*, *blast*)

lemma (**in** *M-basic*) *wf-on-imp-relativized*:
  wf[*A*](*r*) ==> *wellfounded-on*(*M*,*A*,*r*)
apply (*simp add*: *wellfounded-on-def wf-def wf-on-def*, *clarify*)
apply (*drule-tac x*=*x* **in** *spec*, *blast*)
done

lemma (**in** *M-basic*) *wf-imp-relativized*:
  wf(*r*) ==> *wellfounded*(*M*,*r*)
apply (*simp add*: *wellfounded-def wf-def*, *clarify*)
apply (*drule-tac x*=*x* **in** *spec*, *blast*)
done

lemma (**in** *M-basic*) *well-ord-imp-relativized*:
  well-ord(*A*,*r*) ==> *wellordered*(*M*,*A*,*r*)
by (*simp add*: *wellordered-def well-ord-def tot-ord-def part-ord-def*
    linear-imp-relativized trans-on-imp-relativized wf-on-imp-relativized)

The property being well founded (and hence of being well ordered) is not absolute: the set that doesn't contain a minimal element may not exist in the class M. However, every set that is well founded in a transitive model M is well founded (page 124).

3.2 Relativized versions of order-isomorphisms and order types

lemma (**in** *M-basic*) *order-isomorphism-abs* [*simp*]:
  [| *M*(*A*); *M*(*B*); *M*(*f*) |]
  ==> *order-isomorphism*(*M*,*A*,*r*,*B*,*s*,*f*) ⟷ *f* ∈ *ord-iso*(*A*,*r*,*B*,*s*)
by (*simp add*: *order-isomorphism-def ord-iso-def*)

lemma (**in** *M-basic*) *pred-set-abs* [*simp*]:
  [| *M*(*r*); *M*(*B*) |] ==> *pred-set*(*M*,*A*,*x*,*r*,*B*) ⟷ *B* = *Order.pred*(*A*,*x*,*r*)
apply (*simp add*: *pred-set-def Order.pred-def*)
apply (*blast dest*: *transM*)
done

lemma (**in** *M-basic*) *pred-closed* [*intro*,*simp*]:
  [| *M*(*A*); *M*(*r*); *M*(*x*) |] ==> *M*(*Order.pred*(*A*,*x*,*r*))
apply (*simp add*: *Order.pred-def*)
apply (*insert pred-separation* [*of r x*], *simp*)
done

lemma (**in** *M-basic*) *membership-abs* [*simp*]:
  [| *M*(*r*); *M*(*A*) |] ==> *membership*(*M*,*A*,*r*) ⟷ *r* = *Memrel*(*A*)
apply (*simp add*: *membership-def Memrel-def*, *safe*)
 apply (*rule equalityI*)
  apply *clarify*
  apply (*frule transM*, *assumption*)
  apply *blast*
 apply *clarify*
 apply (*subgoal-tac M*(<*xb*,*ya*>), *blast*)
 apply (*blast dest*: *transM*)
apply *auto*
done

lemma (**in** *M-basic*) *M-Memrel-iff*:
  M(*A*) ==>
  Memrel(*A*) = {*z* ∈ *A*∗*A*. ∃ *x*[*M*]. ∃ *y*[*M*]. *z* = ⟨*x*,*y*⟩ & *x* ∈ *y*}
apply (*simp add*: *Memrel-def*)
apply (*blast dest*: *transM*)
done

lemma (**in** *M-basic*) *Memrel-closed* [*intro*,*simp*]:
  M(*A*) ==> *M*(*Memrel*(*A*))
apply (*simp add*: *M-Memrel-iff*)
apply (*insert Memrel-separation*, *simp*)
done

3.3 Main results of Kunen, Chapter 1 section 6

Subset properties– proved outside the locale

lemma *linear-rel-subset*:
  [| *linear-rel*(M,A,r); B<=A |] ==> *linear-rel*(M,B,r)
by (*unfold linear-rel-def*, *blast*)

lemma *transitive-rel-subset*:
  [| *transitive-rel*(M,A,r); B<=A |] ==> *transitive-rel*(M,B,r)
by (*unfold transitive-rel-def*, *blast*)

lemma *wellfounded-on-subset*:
  [| *wellfounded-on*(M,A,r); B<=A |] ==> *wellfounded-on*(M,B,r)
by (*unfold wellfounded-on-def subset-def*, *blast*)

lemma *wellordered-subset*:
  [| *wellordered*(M,A,r); B<=A |] ==> *wellordered*(M,B,r)
apply (*unfold wellordered-def*)
apply (*blast intro*: *linear-rel-subset transitive-rel-subset*
      wellfounded-on-subset)
done

lemma (**in** *M-basic*) *wellfounded-on-asym*:
  [| *wellfounded-on*(M,A,r); <a,x>∈r; a∈A; x∈A; M(A) |] ==> <x,a>∉r
apply (*simp add*: *wellfounded-on-def*)
apply (*drule-tac* x={x,a} **in** *rspec*)
apply (*blast dest*: *transM*)+
done

lemma (**in** *M-basic*) *wellordered-asym*:
  [| *wellordered*(M,A,r); <a,x>∈r; a∈A; x∈A; M(A) |] ==> <x,a>∉r
by (*simp add*: *wellordered-def*, *blast dest*: *wellfounded-on-asym*)

end

4 Relativized Well-Founded Recursion

theory *WFrec* **imports** *Wellorderings* **begin**

4.1 General Lemmas

lemma *apply-recfun2*:
  [| *is-recfun*(r,a,H,f); <x,i>:f |] ==> i = H(x, *restrict*(f,r−``{x}))
apply (*frule apply-recfun*)
apply (*blast dest*: *is-recfun-type fun-is-rel*)
apply (*simp add*: *function-apply-equality* [*OF - is-recfun-imp-function*])
done

Expresses *is-recfun* as a recursion equation

lemma *is-recfun-iff-equation*:
is-recfun$(r,a,H,f) \longleftrightarrow$
$f \in r -``\{a\} \to range(f)$ &
$(\forall x \in r-``\{a\}.\ f`x = H(x,\ restrict(f,\ r-``\{x\})))$
apply (*rule iffI*)
apply (*simp add*: *is-recfun-type apply-recfun Ball-def vimage-singleton-iff*, *clarify*)
apply (*simp add*: *is-recfun-def*)
apply (*rule fun-extension*)
apply *assumption*
apply (*fast intro*: *lam-type*, *simp*)
done

lemma *is-recfun-imp-in-r*: $[|$*is-recfun*$(r,a,H,f);\ \langle x,i\rangle \in f|] ==> \langle x,\ a\rangle \in r$
by (*blast dest*: *is-recfun-type fun-is-rel*)

lemma *trans-Int-eq*:
$[|\ trans(r);\ <y,x> \in r\ |] ==> r -``\{x\} \cap r -``\{y\} = r -``\{y\}$
by (*blast intro*: *transD*)

lemma *is-recfun-restrict-idem*:
is-recfun$(r,a,H,f) ==> restrict(f,\ r -``\{a\}) = f$
apply (*drule is-recfun-type*)
apply (*auto simp add*: *Pi-iff subset-Sigma-imp-relation restrict-idem*)
done

lemma *is-recfun-cong-lemma*:
$[|$ *is-recfun*$(r,a,H,f);\ r = r';\ a = a';\ f = f';$
$!!x\ g.\ [|\ <x,a'> \in r';\ relation(g);\ domain(g) \subseteq r' -``\{x\}\ |]$
$==> H(x,g) = H'(x,g)\ |]$
$==>$ *is-recfun*(r',a',H',f')
apply (*simp add*: *is-recfun-def*)
apply (*erule trans*)
apply (*rule lam-cong*)
apply (*simp-all add*: *vimage-singleton-iff Int-lower2*)
done

For *is-recfun* we need only pay attention to functions whose domains are initial segments of r.

lemma *is-recfun-cong*:
$[|\ r = r';\ a = a';\ f = f';$
$!!x\ g.\ [|\ <x,a'> \in r';\ relation(g);\ domain(g) \subseteq r' -``\{x\}\ |]$
$==> H(x,g) = H'(x,g)\ |]$
$==>$ *is-recfun*$(r,a,H,f) \longleftrightarrow$ *is-recfun*(r',a',H',f')
apply (*rule iffI*)

Messy: fast and blast don't work for some reason

apply (*erule is-recfun-cong-lemma*, *auto*)
apply (*erule is-recfun-cong-lemma*)

apply (*blast intro*: *sym*)+
done

4.2 Reworking of the Recursion Theory Within M

lemma (**in** *M-basic*) *is-recfun-separation'*:
  $[\![\ f \in r -`` \{a\} \to range(f);\ g \in r -`` \{b\} \to range(g);$
   $M(r);\ M(f);\ M(g);\ M(a);\ M(b)\ ]\!]$
  $\Longrightarrow separation(M, \lambda x.\ \neg (\langle x, a\rangle \in r \longrightarrow \langle x, b\rangle \in r \longrightarrow f\ `\ x = g\ `\ x))$
apply (*insert is-recfun-separation* [*of r f g a b*])
apply (*simp add*: *vimage-singleton-iff*)
done

Stated using $trans(r)$ rather than $transitive\text{-}rel(M, A, r)$ because the latter rewrites to the former anyway, by *transitive-rel-abs*. As always, theorems should be expressed in simplified form. The last three M-premises are redundant because of $M(r)$, but without them we'd have to undertake more work to set up the induction formula.

lemma (**in** *M-basic*) *is-recfun-equal* [*rule-format*]:
  $[\![is\text{-}recfun(r,a,H,f);\ \ is\text{-}recfun(r,b,H,g);$
   $wellfounded(M,r);\ \ trans(r);$
   $M(f);\ M(g);\ M(r);\ M(x);\ M(a);\ M(b)\ ]\!]$
  $\Longrightarrow <x,a> \in r \longrightarrow <x,b> \in r \longrightarrow f`x=g`x$
apply (*frule-tac f=f* **in** *is-recfun-type*)
apply (*frule-tac f=g* **in** *is-recfun-type*)
apply (*simp add*: *is-recfun-def*)
apply (*erule-tac a=x* **in** *wellfounded-induct*, *assumption*+)

Separation to justify the induction

apply (*blast intro*: *is-recfun-separation'*)

Now the inductive argument itself

apply *clarify*
apply (*erule ssubst*)+
apply (*simp* (*no-asm-simp*) *add*: *vimage-singleton-iff restrict-def*)
apply (*rename-tac x1*)
apply (*rule-tac t=%z. H*(*x1*,*z*) **in** *subst-context*)
apply (*subgoal-tac* $\forall y \in r-``\{x1\}.\ \forall z.\ <y,z>\in f \longleftrightarrow <y,z>\in g$)
apply (*blast intro*: *transD*)
apply (*simp add*: *apply-iff*)
apply (*blast intro*: *transD sym*)
done

lemma (**in** *M-basic*) *is-recfun-cut*:
  $[\![is\text{-}recfun(r,a,H,f);\ \ is\text{-}recfun(r,b,H,g);$
   $wellfounded(M,r);\ trans(r);$
   $M(f);\ M(g);\ M(r);\ <b,a> \in r\ ]\!]$
  $\Longrightarrow restrict(f, r-``\{b\}) = g$
apply (*frule-tac f=f* **in** *is-recfun-type*)

apply (*rule fun-extension*)
apply (*blast intro*: *transD restrict-type2*)
apply (*erule is-recfun-type*, *simp*)
apply (*blast intro*: *is-recfun-equal transD dest*: *transM*)
done

lemma (**in** *M-basic*) *is-recfun-functional*:
[|*is-recfun*(*r*,*a*,*H*,*f*); *is-recfun*(*r*,*a*,*H*,*g*);
wellfounded(*M*,*r*); *trans*(*r*); *M*(*f*); *M*(*g*); *M*(*r*) |] ==> *f*=*g*
apply (*rule fun-extension*)
apply (*erule is-recfun-type*)+
apply (*blast intro*!: *is-recfun-equal dest*: *transM*)
done

Tells us that *is-recfun* can (in principle) be relativized.

lemma (**in** *M-basic*) *is-recfun-relativize*:
[| *M*(*r*); *M*(*f*); ∀ *x*[*M*]. ∀ *g*[*M*]. *function*(*g*) ⟶ *M*(*H*(*x*,*g*)) |]
==> *is-recfun*(*r*,*a*,*H*,*f*) ⟷
(∀ *z*[*M*]. *z* ∈ *f* ⟷
(∃ *x*[*M*]. <*x*,*a*> ∈ *r* & *z* = <*x*, *H*(*x*, *restrict*(*f*, *r*−‘‘{*x*}))>))
apply (*simp add*: *is-recfun-def lam-def*)
apply (*safe intro*!: *equalityI*)
apply (*drule equalityD1* [*THEN subsetD*], *assumption*)
apply (*blast dest*: *pair-components-in-M*)
apply (*blast elim*!: *equalityE dest*: *pair-components-in-M*)
apply (*frule transM*, *assumption*)
apply *simp*
apply *blast*
apply (*subgoal-tac is-function*(*M*,*f*))

We use *is-function* rather than *function* because the subgoal's easier to prove with relativized quantifiers!

prefer *2* **apply** (*simp add*: *is-function-def*)
apply (*frule pair-components-in-M*, *assumption*)
apply (*simp add*: *is-recfun-imp-function function-restrictI*)
done

lemma (**in** *M-basic*) *is-recfun-restrict*:
[| *wellfounded*(*M*,*r*); *trans*(*r*); *is-recfun*(*r*,*x*,*H*,*f*); ⟨*y*,*x*⟩ ∈ *r*;
M(*r*); *M*(*f*);
∀ *x*[*M*]. ∀ *g*[*M*]. *function*(*g*) ⟶ *M*(*H*(*x*,*g*)) |]
==> *is-recfun*(*r*, *y*, *H*, *restrict*(*f*, *r* −‘‘ {*y*}))
apply (*frule pair-components-in-M*, *assumption*, *clarify*)
apply (*simp* (*no-asm-simp*) *add*: *is-recfun-relativize restrict-iff*
trans-Int-eq)
apply *safe*
apply (*simp-all add*: *vimage-singleton-iff is-recfun-type* [*THEN apply-iff*])
apply (*frule-tac x*=*xa* **in** *pair-components-in-M*, *assumption*)
apply (*frule-tac x*=*xa* **in** *apply-recfun*, *blast intro*: *transD*)

apply (*simp add*: *is-recfun-type* [*THEN apply-iff*]
is-recfun-imp-function function-restrictI)
apply (*blast intro*: *apply-recfun dest*: *transD*)
done

lemma (**in** *M-basic*) *restrict-Y-lemma*:
[| *wellfounded*(*M*,*r*); *trans*(*r*); *M*(*r*);
∀ *x*[*M*]. ∀ *g*[*M*]. *function*(*g*) ⟶ *M*(*H*(*x*,*g*)); *M*(*Y*);
∀ *b*[*M*].
b ∈ *Y* ⟷
(∃ *x*[*M*]. <*x*,*a1*> ∈ *r* &
(∃ *y*[*M*]. *b* = ⟨*x*,*y*⟩ & (∃ *g*[*M*]. *is-recfun*(*r*,*x*,*H*,*g*) ∧ *y* = *H*(*x*,*g*))));
⟨*x*,*a1*⟩ ∈ *r*; *is-recfun*(*r*,*x*,*H*,*f*); *M*(*f*) |]
==> *restrict*(*Y*, *r* −‘‘ {*x*}) = *f*
apply (*subgoal-tac* ∀ *y* ∈ *r*−‘‘{*x*}. ∀ *z*. <*y*,*z*>:*Y* ⟷ <*y*,*z*>:*f*)
apply (*simp* (*no-asm-simp*) *add*: *restrict-def*)
apply (*thin-tac rall*(*M*,*P*) **for** *P*)+ — essential for efficiency
apply (*frule is-recfun-type* [*THEN fun-is-rel*], *blast*)
apply (*frule pair-components-in-M*, *assumption*, *clarify*)
apply (*rule iffI*)
apply (*frule-tac y*=<*y*,*z*> **in** *transM*, *assumption*)
apply (*clarsimp simp add*: *vimage-singleton-iff is-recfun-type* [*THEN apply-iff*]
apply-recfun is-recfun-cut)

Opposite inclusion: something in f, show in Y

apply (*frule-tac y*=<*y*,*z*> **in** *transM*, *assumption*)
apply (*simp add*: *vimage-singleton-iff*)
apply (*rule conjI*)
apply (*blast dest*: *transD*)
apply (*rule-tac x*=*restrict*(*f*, *r* −‘‘ {*y*}) **in** *rexI*)
apply (*simp-all add*: *is-recfun-restrict*
apply-recfun is-recfun-type [*THEN apply-iff*])
done

For typical applications of Replacement for recursive definitions

lemma (**in** *M-basic*) *univalent-is-recfun*:
[|*wellfounded*(*M*,*r*); *trans*(*r*); *M*(*r*)|]
==> *univalent* (*M*, *A*, λ*x p*.
∃ *y*[*M*]. *p* = ⟨*x*,*y*⟩ & (∃ *f*[*M*]. *is-recfun*(*r*,*x*,*H*,*f*) & *y* = *H*(*x*,*f*)))
apply (*simp add*: *univalent-def*)
apply (*blast dest*: *is-recfun-functional*)
done

Proof of the inductive step for *exists-is-recfun*, since we must prove two versions.

lemma (**in** *M-basic*) *exists-is-recfun-indstep*:
[|∀ *y*. ⟨*y*, *a1*⟩ ∈ *r* ⟶ (∃ *f*[*M*]. *is-recfun*(*r*, *y*, *H*, *f*));
wellfounded(*M*,*r*); *trans*(*r*); *M*(*r*); *M*(*a1*);
strong-replacement(*M*, λ*x z*.

$\exists y[M]. \exists g[M]. pair(M,x,y,z)$ & $is\text{-}recfun(r,x,H,g)$ & $y = H(x,g))$;
$\forall x[M]. \forall g[M]. function(g) \longrightarrow M(H(x,g))$|]
==> $\exists f[M]. is\text{-}recfun(r,a1,H,f)$
apply (*drule-tac A=r−‘‘{a1}* **in** *strong-replacementD*)
apply *blast*

Discharge the "univalent" obligation of Replacement

apply (*simp add*: *univalent-is-recfun*)

Show that the constructed object satisfies *is-recfun*

apply *clarify*
apply (*rule-tac x=Y* **in** *rexI*)

Unfold only the top-level occurrence of *is-recfun*

apply (*simp* (*no-asm-simp*) *add*: *is-recfun-relativize* [*of concl*: - *a1*])

The big iff-formula defining Y is now redundant

apply *safe*
apply (*simp add*: *vimage-singleton-iff restrict-Y-lemma* [*of r H* - *a1*])

one more case

apply (*simp* (*no-asm-simp*) *add*: *Bex-def vimage-singleton-iff*)
apply (*drule-tac x1=x* **in** *spec* [*THEN mp*], *assumption*, *clarify*)
apply (*rename-tac f*)
apply (*rule-tac x=f* **in** *rexI*)
apply (*simp-all add*: *restrict-Y-lemma* [*of r H*])

FIXME: should not be needed!

apply (*subst restrict-Y-lemma* [*of r H*])
apply (*simp add*: *vimage-singleton-iff*)+
apply *blast*+
done

Relativized version, when we have the (currently weaker) premise *wellfounded*(M, r)

lemma (**in** *M-basic*) *wellfounded-exists-is-recfun*:
[|*wellfounded*(M,r); *trans*(r);
separation$(M, \lambda x. \sim (\exists f[M]. is\text{-}recfun(r, x, H, f)))$;
strong-replacement$(M, \lambda x\ z.$
$\exists y[M]. \exists g[M]. pair(M,x,y,z)$ & $is\text{-}recfun(r,x,H,g)$ & $y = H(x,g))$;
$M(r)$; $M(a)$;
$\forall x[M]. \forall g[M]. function(g) \longrightarrow M(H(x,g))$ |]
==> $\exists f[M]. is\text{-}recfun(r,a,H,f)$
apply (*rule wellfounded-induct*, *assumption*+, *clarify*)
apply (*rule exists-is-recfun-indstep*, *assumption*+)
done

lemma (**in** *M-basic*) *wf-exists-is-recfun* [*rule-format*]:

```
  [|wf(r); trans(r); M(r);
     strong-replacement(M, λx z.
       ∃ y[M]. ∃ g[M]. pair(M,x,y,z) & is-recfun(r,x,H,g) & y = H(x,g));
     ∀ x[M]. ∀ g[M]. function(g) ⟶ M(H(x,g)) |]
    ==> M(a) ⟶ (∃ f[M]. is-recfun(r,a,H,f))
apply (rule wf-induct, assumption+)
apply (frule wf-imp-relativized)
apply (intro impI)
apply (rule exists-is-recfun-indstep)
     apply (blast dest: transM del: rev-rallE, assumption+)
done
```

4.3 Relativization of the ZF Predicate *is-recfun*

```
definition
  M-is-recfun :: [i=>o, [i,i,i]=>o, i, i, i] => o where
   M-is-recfun(M,MH,r,a,f) ==
     ∀ z[M]. z ∈ f ⟷
          (∃ x[M]. ∃ y[M]. ∃ xa[M]. ∃ sx[M]. ∃ r-sx[M]. ∃ f-r-sx[M].
             pair(M,x,y,z) & pair(M,x,a,xa) & upair(M,x,x,sx) &
             pre-image(M,r,sx,r-sx) & restriction(M,f,r-sx,f-r-sx) &
             xa ∈ r & MH(x, f-r-sx, y))

definition
  is-wfrec :: [i=>o, [i,i,i]=>o, i, i, i] => o where
   is-wfrec(M,MH,r,a,z) ==
     ∃ f[M]. M-is-recfun(M,MH,r,a,f) & MH(a,f,z)

definition
  wfrec-replacement :: [i=>o, [i,i,i]=>o, i] => o where
   wfrec-replacement(M,MH,r) ==
       strong-replacement(M,
            λx z. ∃ y[M]. pair(M,x,y,z) & is-wfrec(M,MH,r,x,y))

lemma (in M-basic) is-recfun-abs:
     [| ∀ x[M]. ∀ g[M]. function(g) ⟶ M(H(x,g)); M(r); M(a); M(f);
        relation2(M,MH,H) |]
      ==> M-is-recfun(M,MH,r,a,f) ⟷ is-recfun(r,a,H,f)
apply (simp add: M-is-recfun-def relation2-def is-recfun-relativize)
apply (rule rall-cong)
apply (blast dest: transM)
done

lemma M-is-recfun-cong [cong]:
     [| r = r'; a = a'; f = f';
       !!x g y. [| M(x); M(g); M(y) |] ==> MH(x,g,y) ⟷ MH'(x,g,y) |]
      ==> M-is-recfun(M,MH,r,a,f) ⟷ M-is-recfun(M,MH',r',a',f')
by (simp add: M-is-recfun-def)
```

lemma (**in** *M-basic*) *is-wfrec-abs*:
[| ∀ x[M]. ∀ g[M]. function(g) ⟶ M(H(x,g));
relation2(M,MH,H); M(r); M(a); M(z) |]
==> is-wfrec(M,MH,r,a,z) ⟷
(∃ g[M]. is-recfun(r,a,H,g) & z = H(a,g))
by (*simp add*: *is-wfrec-def relation2-def is-recfun-abs*)

Relating *wfrec-replacement* to native constructs

lemma (**in** *M-basic*) *wfrec-replacement′*:
[|wfrec-replacement(M,MH,r);
∀ x[M]. ∀ g[M]. function(g) ⟶ M(H(x,g));
relation2(M,MH,H); M(r)|]
==> strong-replacement(M, λx z. ∃ y[M].
pair(M,x,y,z) & (∃ g[M]. is-recfun(r,x,H,g) & y = H(x,g)))
by (*simp add*: *wfrec-replacement-def is-wfrec-abs*)

lemma *wfrec-replacement-cong* [*cong*]:
[| !!x y z. [| M(x); M(y); M(z) |] ==> MH(x,y,z) ⟷ MH′(x,y,z);
r=r′ |]
==> wfrec-replacement(M, %x y. MH(x,y), r) ⟷
wfrec-replacement(M, %x y. MH′(x,y), r′)
by (*simp add*: *is-wfrec-def wfrec-replacement-def*)

end

5 Absoluteness of Well-Founded Recursion

theory *WF-absolute* **imports** *WFrec* **begin**

5.1 Transitive closure without fixedpoints

definition
rtrancl-alt :: [i,i]=>i **where**
rtrancl-alt(A,r) ==
{p ∈ A*A. ∃ n∈nat. ∃ f ∈ succ(n) −> A.
(∃ x y. p = <x,y> & f‘0 = x & f‘n = y) &
(∀ i∈n. <f‘i, f‘succ(i)> ∈ r)}

lemma *alt-rtrancl-lemma1* [*rule-format*]:
n ∈ nat
==> ∀ f ∈ succ(n) −> field(r).
(∀ i∈n. ⟨f‘i, f ‘ succ(i)⟩ ∈ r) ⟶ ⟨f‘0, f‘n⟩ ∈ r^*
apply (*induct-tac n*)
apply (*simp-all add*: *apply-funtype rtrancl-refl*, *clarify*)
apply (*rename-tac n f*)
apply (*rule rtrancl-into-rtrancl*)
prefer *2* **apply** *assumption*
apply (*drule-tac x=restrict(f,succ(n))* **in** *bspec*)

apply (*blast intro*: *restrict-type2*)
apply (*simp add*: *Ord-succ-mem-iff nat-0-le* [*THEN ltD*] *leI* [*THEN ltD*] *ltI*)
done

lemma *rtrancl-alt-subset-rtrancl*: *rtrancl-alt*(*field*(*r*),*r*) ⊆ *r*^*
apply (*simp add*: *rtrancl-alt-def*)
apply (*blast intro*: *alt-rtrancl-lemma1*)
done

lemma *rtrancl-subset-rtrancl-alt*: *r*^* ⊆ *rtrancl-alt*(*field*(*r*),*r*)
apply (*simp add*: *rtrancl-alt-def*, *clarify*)
apply (*frule rtrancl-type* [*THEN subsetD*], *clarify*, *simp*)
apply (*erule rtrancl-induct*)

Base case, trivial

apply (*rule-tac x*=*0* **in** *bexI*)
apply (*rule-tac x*=λ*x*∈*1*. *xa* **in** *bexI*)
apply *simp-all*

Inductive step

apply *clarify*
apply (*rename-tac n f*)
apply (*rule-tac x*=*succ*(*n*) **in** *bexI*)
apply (*rule-tac x*=λ*i*∈*succ*(*succ*(*n*)). *if i*=*succ*(*n*) *then z else f*'*i* **in** *bexI*)
apply (*simp add*: *Ord-succ-mem-iff nat-0-le* [*THEN ltD*] *leI* [*THEN ltD*] *ltI*)
apply (*blast intro*: *mem-asym*)
apply *typecheck*
apply *auto*
done

lemma *rtrancl-alt-eq-rtrancl*: *rtrancl-alt*(*field*(*r*),*r*) = *r*^*
by (*blast del*: *subsetI*
intro: *rtrancl-alt-subset-rtrancl rtrancl-subset-rtrancl-alt*)

definition
rtran-closure-mem :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
— The property of belonging to *rtran-closure*(*r*)
rtran-closure-mem(*M*,*A*,*r*,*p*) ==
∃ *nnat*[*M*]. ∃ *n*[*M*]. ∃ *n'*[*M*].
omega(*M*,*nnat*) & *n*∈*nnat* & *successor*(*M*,*n*,*n'*) &
(∃ *f*[*M*]. *typed-function*(*M*,*n'*,*A*,*f*) &
(∃ *x*[*M*]. ∃ *y*[*M*]. ∃ *zero*[*M*]. *pair*(*M*,*x*,*y*,*p*) & *empty*(*M*,*zero*) &
fun-apply(*M*,*f*,*zero*,*x*) & *fun-apply*(*M*,*f*,*n*,*y*)) &
(∀ *j*[*M*]. *j*∈*n* ⟶
(∃ *fj*[*M*]. ∃ *sj*[*M*]. ∃ *fsj*[*M*]. ∃ *ffp*[*M*].
fun-apply(*M*,*f*,*j*,*fj*) & *successor*(*M*,*j*,*sj*) &
fun-apply(*M*,*f*,*sj*,*fsj*) & *pair*(*M*,*fj*,*fsj*,*ffp*) & *ffp* ∈ *r*)))

```
definition
  rtran-closure :: [i=>o,i,i] => o where
    rtran-closure(M,r,s) ==
        ∀ A[M]. is-field(M,r,A) ⟶
         (∀ p[M]. p ∈ s ⟷ rtran-closure-mem(M,A,r,p))

definition
  tran-closure :: [i=>o,i,i] => o where
    tran-closure(M,r,t) ==
         ∃ s[M]. rtran-closure(M,r,s) & composition(M,r,s,t)

locale M-trancl = M-basic +
  assumes rtrancl-separation:
         [| M(r); M(A) |] ==> separation (M, rtran-closure-mem(M,A,r))
      and wellfounded-trancl-separation:
         [| M(r); M(Z) |] ==>
          separation (M, λx.
              ∃ w[M]. ∃ wx[M]. ∃ rp[M].
               w ∈ Z & pair(M,w,x,wx) & tran-closure(M,r,rp) & wx ∈ rp)
      and M-nat [iff] : M(nat)

lemma (in M-trancl) rtran-closure-mem-iff:
     [|M(A); M(r); M(p)|]
      ==> rtran-closure-mem(M,A,r,p) ⟷
          (∃ n[M]. n∈nat &
           (∃ f[M]. f ∈ succ(n) -> A &
            (∃ x[M]. ∃ y[M]. p = <x,y> & f`0 = x & f`n = y) &
                           (∀ i∈n. <f`i, f`succ(i)> ∈ r)))
  apply (simp add: rtran-closure-mem-def Ord-succ-mem-iff nat-0-le [THEN ltD]
M-nat)
done

lemma (in M-trancl) rtran-closure-rtrancl:
     M(r) ==> rtran-closure(M,r,rtrancl(r))
apply (simp add: rtran-closure-def rtran-closure-mem-iff
                 rtrancl-alt-eq-rtrancl [symmetric] rtrancl-alt-def)
apply (auto simp add: nat-0-le [THEN ltD] apply-funtype)
done

lemma (in M-trancl) rtrancl-closed [intro,simp]:
     M(r) ==> M(rtrancl(r))
apply (insert rtrancl-separation [of r field(r)])
apply (simp add: rtrancl-alt-eq-rtrancl [symmetric]
                 rtrancl-alt-def rtran-closure-mem-iff M-nat)
done

lemma (in M-trancl) rtrancl-abs [simp]:
     [| M(r); M(z) |] ==> rtran-closure(M,r,z) ⟷ z = rtrancl(r)
apply (rule iffI)
```

Proving the right-to-left implication

prefer *2* **apply** (*blast intro*: *rtran-closure-rtrancl*)
apply (*rule M-equalityI*)
apply (*simp add*: *rtran-closure-def rtrancl-alt-eq-rtrancl* [*symmetric*]
rtrancl-alt-def rtran-closure-mem-iff)
apply (*auto simp add*: *nat-0-le* [*THEN ltD*] *apply-funtype*)
done

lemma (**in** *M-trancl*) *trancl-closed* [*intro*,*simp*]:
$M(r)$ ==> $M(trancl(r))$
by (*simp add*: *trancl-def*)

lemma (**in** *M-trancl*) *trancl-abs* [*simp*]:
[| $M(r)$; $M(z)$ |] ==> *tran-closure*$(M,r,z) \longleftrightarrow z = trancl(r)$
by (*simp add*: *tran-closure-def trancl-def*)

lemma (**in** *M-trancl*) *wellfounded-trancl-separation'*:
[| $M(r)$; $M(Z)$ |] ==> *separation* $(M, \lambda x. \exists w[M]. w \in Z$ & <w,x> $\in r\hat{}+)$
by (*insert wellfounded-trancl-separation* [*of r Z*], *simp*)

Alternative proof of *wf-on-trancl*; inspiration for the relativized version. Original version is on theory WF.

lemma [| $wf[A](r)$; $r-``A \subseteq A$ |] ==> $wf[A](r\hat{}+)$
apply (*simp add*: *wf-on-def wf-def*)
apply (*safe*)
apply (*drule-tac* $x = \{x\in A.\ \exists w.\ \langle w,x\rangle \in r\hat{}+$ & $w \in Z\}$ **in** *spec*)
apply (*blast elim*: *tranclE*)
done

lemma (**in** *M-trancl*) *wellfounded-on-trancl*:
[| *wellfounded-on*(M,A,r); $r-``A \subseteq A$; $M(r)$; $M(A)$ |]
==> *wellfounded-on*$(M,A,r\hat{}+)$
apply (*simp add*: *wellfounded-on-def*)
apply (*safe intro*!: *equalityI*)
apply (*rename-tac Z x*)
apply (*subgoal-tac* $M(\{x\in A.\ \exists w[M].\ w \in Z$ & $\langle w,x\rangle \in r\hat{}+\}))$
prefer *2*
apply (*blast intro*: *wellfounded-trancl-separation'*)
apply (*drule-tac* $x = \{x\in A.\ \exists w[M].\ w \in Z$ & $\langle w,x\rangle \in r\hat{}+\}$ **in** *rspec*, *safe*)
apply (*blast dest*: *transM*, *simp*)
apply (*rename-tac y w*)
apply (*drule-tac x=w* **in** *bspec*, *assumption*, *clarify*)
apply (*erule tranclE*)
apply (*blast dest*: *transM*)
apply *blast*
done

lemma (**in** *M-trancl*) *wellfounded-trancl*:
[|*wellfounded*(M,r); $M(r)$|] ==> *wellfounded*$(M,r\hat{}+)$

apply (*simp add*: *wellfounded-iff-wellfounded-on-field*)
apply (*rule wellfounded-on-subset-A*, *erule wellfounded-on-trancl*)
  apply *blast*
 apply (*simp-all add*: *trancl-type* [*THEN field-rel-subset*])
done

Absoluteness for wfrec-defined functions.

lemma (**in** *M-trancl*) *wfrec-relativize*:
 [|*wf*(*r*); *M*(*a*); *M*(*r*);
   strong-replacement(*M*, $\lambda x\ z.\ \exists\, y[M].\ \exists\, g[M].$
     pair(*M*,*x*,*y*,*z*) &
     is-recfun($r\hat{\ }+$, *x*, $\lambda x\, f.$ *H*(*x*, *restrict*(*f*, *r* −‘‘ {*x*})), *g*) &
     y = *H*(*x*, *restrict*(*g*, *r* −‘‘ {*x*})));
   $\forall\, x[M].\ \forall\, g[M].$ *function*(*g*) $\longrightarrow$ *M*(*H*(*x*,*g*))|]
 ==> *wfrec*(*r*,*a*,*H*) = *z* $\longleftrightarrow$
   ($\exists\, f[M]$. *is-recfun*($r\hat{\ }+$, *a*, $\lambda x\, f.$ *H*(*x*, *restrict*(*f*, *r* −‘‘ {*x*})), *f*) &
     z = *H*(*a*,*restrict*(*f*,*r*−‘‘{*a*})))
apply (*frule wf-trancl*)
apply (*simp add*: *wftrec-def wfrec-def*, *safe*)
 apply (*frule wf-exists-is-recfun*
       [*of concl*: $r\hat{\ }+$ *a* $\lambda x\, f.$ *H*(*x*, *restrict*(*f*, *r* −‘‘ {*x*}))])
   apply (*simp-all add*: *trans-trancl function-restrictI trancl-subset-times*)
 apply (*clarify*, *rule-tac x*=*x* **in** *rexI*)
 apply (*simp-all add*: *the-recfun-eq trans-trancl trancl-subset-times*)
done

Assuming *r* is transitive simplifies the occurrences of *H*. The premise *relation*(*r*) is necessary before we can replace $r\hat{\ }+$ by *r*.

theorem (**in** *M-trancl*) *trans-wfrec-relativize*:
 [|*wf*(*r*); *trans*(*r*); *relation*(*r*); *M*(*r*); *M*(*a*);
   wfrec-replacement(*M*,*MH*,*r*); *relation2*(*M*,*MH*,*H*);
   $\forall\, x[M].\ \forall\, g[M].$ *function*(*g*) $\longrightarrow$ *M*(*H*(*x*,*g*))|]
 ==> *wfrec*(*r*,*a*,*H*) = *z* $\longleftrightarrow$ ($\exists\, f[M]$. *is-recfun*(*r*,*a*,*H*,*f*) & *z* = *H*(*a*,*f*))
apply (*frule wfrec-replacement′*, *assumption*+)
apply (*simp cong*: *is-recfun-cong*
    add: *wfrec-relativize trancl-eq-r*
      is-recfun-restrict-idem domain-restrict-idem)
done

theorem (**in** *M-trancl*) *trans-wfrec-abs*:
 [|*wf*(*r*); *trans*(*r*); *relation*(*r*); *M*(*r*); *M*(*a*); *M*(*z*);
   wfrec-replacement(*M*,*MH*,*r*); *relation2*(*M*,*MH*,*H*);
   $\forall\, x[M].\ \forall\, g[M].$ *function*(*g*) $\longrightarrow$ *M*(*H*(*x*,*g*))|]
 ==> *is-wfrec*(*M*,*MH*,*r*,*a*,*z*) $\longleftrightarrow$ *z*=*wfrec*(*r*,*a*,*H*)
by (*simp add*: *trans-wfrec-relativize* [*THEN iff-sym*] *is-wfrec-abs*, *blast*)

lemma (**in** *M-trancl*) *trans-eq-pair-wfrec-iff*:
 [|*wf*(*r*); *trans*(*r*); *relation*(*r*); *M*(*r*); *M*(*y*);

wfrec-replacement(*M*,*MH*,*r*); *relation2*(*M*,*MH*,*H*);
∀ *x*[*M*]. ∀ *g*[*M*]. *function*(*g*) ⟶ *M*(*H*(*x*,*g*))|]
==> *y* = <*x*, *wfrec*(*r*, *x*, *H*)> ⟷
(∃ *f*[*M*]. *is-recfun*(*r*,*x*,*H*,*f*) & *y* = <*x*, *H*(*x*,*f*)>)
apply *safe*
apply (*simp add*: *trans-wfrec-relativize* [*THEN iff-sym*, *of concl*: - *x*])

converse direction

apply (*rule sym*)
apply (*simp add*: *trans-wfrec-relativize*, *blast*)
done

5.2 M is closed under well-founded recursion

Lemma with the awkward premise mentioning *wfrec*.

lemma (**in** *M-trancl*) *wfrec-closed-lemma* [*rule-format*]:
[|*wf*(*r*); *M*(*r*);
strong-replacement(*M*, λ*x y*. *y* = ⟨*x*, *wfrec*(*r*, *x*, *H*)⟩);
∀ *x*[*M*]. ∀ *g*[*M*]. *function*(*g*) ⟶ *M*(*H*(*x*,*g*)) |]
==> *M*(*a*) ⟶ *M*(*wfrec*(*r*,*a*,*H*))
apply (*rule-tac a*=*a* **in** *wf-induct*, *assumption*+)
apply (*subst wfrec*, *assumption*, *clarify*)
apply (*drule-tac x1*=*x* **and** *x*=λ*x*∈*r* −‘‘ {*x*}. *wfrec*(*r*, *x*, *H*)
in *rspec* [*THEN rspec*])
apply (*simp-all add*: *function-lam*)
apply (*blast intro*: *lam-closed dest*: *pair-components-in-M*)
done

Eliminates one instance of replacement.

lemma (**in** *M-trancl*) *wfrec-replacement-iff*:
strong-replacement(*M*, λ*x z*.
∃ *y*[*M*]. *pair*(*M*,*x*,*y*,*z*) & (∃ *g*[*M*]. *is-recfun*(*r*,*x*,*H*,*g*) & *y* = *H*(*x*,*g*))) ⟷
strong-replacement(*M*,
λ*x y*. ∃ *f*[*M*]. *is-recfun*(*r*,*x*,*H*,*f*) & *y* = <*x*, *H*(*x*,*f*)>)
apply *simp*
apply (*rule strong-replacement-cong*, *blast*)
done

Useful version for transitive relations

theorem (**in** *M-trancl*) *trans-wfrec-closed*:
[|*wf*(*r*); *trans*(*r*); *relation*(*r*); *M*(*r*); *M*(*a*);
wfrec-replacement(*M*,*MH*,*r*); *relation2*(*M*,*MH*,*H*);
∀ *x*[*M*]. ∀ *g*[*M*]. *function*(*g*) ⟶ *M*(*H*(*x*,*g*)) |]
==> *M*(*wfrec*(*r*,*a*,*H*))
apply (*frule wfrec-replacement*′, *assumption*+)
apply (*frule wfrec-replacement-iff* [*THEN iffD1*])
apply (*rule wfrec-closed-lemma*, *assumption*+)
apply (*simp-all add*: *wfrec-replacement-iff trans-eq-pair-wfrec-iff*)
done

5.3 Absoluteness without assuming transitivity

```
lemma (in M-trancl) eq-pair-wfrec-iff:
  [|wf(r);  M(r);  M(y);
    strong-replacement(M, λx z. ∃ y[M]. ∃ g[M].
        pair(M,x,y,z) &
        is-recfun(r^+, x, λx f. H(x, restrict(f, r −`` {x})), g) &
        y = H(x, restrict(g, r −`` {x})));
    ∀ x[M]. ∀ g[M]. function(g) ⟶ M(H(x,g))|]
  ==> y = <x, wfrec(r, x, H)> ⟷
      (∃ f[M]. is-recfun(r^+, x, λx f. H(x, restrict(f, r −`` {x})), f) &
          y = <x, H(x,restrict(f,r−``{x}))>)
apply safe
 apply (simp add: wfrec-relativize [THEN iff-sym, of concl: - x])
```

converse direction

```
apply (rule sym)
apply (simp add: wfrec-relativize, blast)
done
```

Full version not assuming transitivity, but maybe not very useful.

```
theorem (in M-trancl) wfrec-closed:
    [|wf(r); M(r); M(a);
      wfrec-replacement(M,MH,r^+);
      relation2(M,MH, λx f. H(x, restrict(f, r −`` {x})));
      ∀ x[M]. ∀ g[M]. function(g) ⟶ M(H(x,g)) |]
    ==> M(wfrec(r,a,H))
apply (frule wfrec-replacement'
               [of MH r^+ λx f. H(x, restrict(f, r −`` {x}))])
  prefer 4
  apply (frule wfrec-replacement-iff [THEN iffD1])
  apply (rule wfrec-closed-lemma, assumption+)
   apply (simp-all add: eq-pair-wfrec-iff func.function-restrictI)
done

end
```

6 Absoluteness Properties for Recursive Datatypes

```
theory Datatype-absolute
  imports
   ~~/src/ZF/Constructible/Formula
   WF-absolute
begin
```

6.1 The lfp of a continuous function can be expressed as a union

```
definition
```

directed :: *i*=>*o* **where**
directed(*A*) == *A*≠*0* & (∀ *x*∈*A*. ∀ *y*∈*A*. *x* ∪ *y* ∈ *A*)

definition
contin :: (*i*=>*i*) => *o* **where**
contin(*h*) == (∀ *A*. *directed*(*A*) ⟶ *h*(⋃ *A*) = (⋃ *X*∈*A*. *h*(*X*)))

lemma *bnd-mono-iterates-subset*: [|*bnd-mono*(*D*, *h*); *n* ∈ *nat*|] ==> *h* ^*n* (*0*) ⊆ *D*
apply (*induct-tac n*)
apply (*simp-all add*: *bnd-mono-def*, *blast*)
done

lemma *bnd-mono-increasing* [*rule-format*]:
[|*i* ∈ *nat*; *j* ∈ *nat*; *bnd-mono*(*D*,*h*)|] ==> *i* ≤ *j* ⟶ *h* ^*i*(*0*) ⊆ *h* ^*j*(*0*)
apply (*rule-tac m*=*i* **and** *n*=*j* **in** *diff-induct*, *simp-all*)
apply (*blast del*: *subsetI*
intro: *bnd-mono-iterates-subset bnd-monoD2* [*of concl*: *h*])
done

lemma *directed-iterates*: *bnd-mono*(*D*,*h*) ==> *directed*({*h* ^*n* (*0*). *n*∈*nat*})
apply (*simp add*: *directed-def*, *clarify*)
apply (*rename-tac i j*)
apply (*rule-tac x*=*i* ∪ *j* **in** *bexI*)
apply (*rule-tac i* = *i* **and** *j* = *j* **in** *Ord-linear-le*)
apply (*simp-all add*: *subset-Un-iff* [*THEN iffD1*] *le-imp-subset*
subset-Un-iff2 [*THEN iffD1*])
apply (*simp-all add*: *subset-Un-iff* [*THEN iff-sym*] *bnd-mono-increasing*
subset-Un-iff2 [*THEN iff-sym*])
done

lemma *contin-iterates-eq*:
[|*bnd-mono*(*D*, *h*); *contin*(*h*)|]
==> *h*(⋃ *n*∈*nat*. *h* ^*n* (*0*)) = (⋃ *n*∈*nat*. *h* ^*n* (*0*))
apply (*simp add*: *contin-def directed-iterates*)
apply (*rule trans*)
apply (*rule equalityI*)
apply (*simp-all add*: *UN-subset-iff*)
apply *safe*
apply (*erule-tac* [*2*] *natE*)
apply (*rule-tac a*=*succ*(*x*) **in** *UN-I*)
apply *simp-all*
apply *blast*
done

lemma *lfp-subset-Union*:
[|*bnd-mono*(*D*, *h*); *contin*(*h*)|] ==> *lfp*(*D*,*h*) ⊆ (⋃ *n*∈*nat*. *h* ^*n*(*0*))
apply (*rule lfp-lowerbound*)
apply (*simp add*: *contin-iterates-eq*)

apply (*simp add*: *contin-def bnd-mono-iterates-subset UN-subset-iff*)
done

lemma *Union-subset-lfp*:
bnd-mono(*D*,*h*) ==> ($\bigcup$ *n*∈*nat*. *h*^*n*(*0*)) ⊆ *lfp*(*D*,*h*)
apply (*simp add*: *UN-subset-iff*)
apply (*rule ballI*)
apply (*induct-tac n*, *simp-all*)
apply (*rule subset-trans* [*of - h*(*lfp*(*D*,*h*))])
apply (*blast dest*: *bnd-monoD2* [*OF - - lfp-subset*])
apply (*erule lfp-lemma2*)
done

lemma *lfp-eq-Union*:
[|*bnd-mono*(*D*, *h*); *contin*(*h*)|] ==> *lfp*(*D*,*h*) = ($\bigcup$ *n*∈*nat*. *h*^*n*(*0*))
by (*blast del*: *subsetI*
intro: *lfp-subset-Union Union-subset-lfp*)

6.1.1 Some Standard Datatype Constructions Preserve Continuity

lemma *contin-imp-mono*: [|*X*⊆*Y*; *contin*(*F*)|] ==> *F*(*X*) ⊆ *F*(*Y*)
apply (*simp add*: *contin-def*)
apply (*drule-tac x*={*X*,*Y*} **in** *spec*)
apply (*simp add*: *directed-def subset-Un-iff2 Un-commute*)
done

lemma *sum-contin*: [|*contin*(*F*); *contin*(*G*)|] ==> *contin*(λ*X*. *F*(*X*) + *G*(*X*))
by (*simp add*: *contin-def*, *blast*)

lemma *prod-contin*: [|*contin*(*F*); *contin*(*G*)|] ==> *contin*(λ*X*. *F*(*X*) ∗ *G*(*X*))
apply (*subgoal-tac* ∀ *B C*. *F*(*B*) ⊆ *F*(*B* ∪ *C*))
prefer *2* **apply** (*simp add*: *Un-upper1 contin-imp-mono*)
apply (*subgoal-tac* ∀ *B C*. *G*(*C*) ⊆ *G*(*B* ∪ *C*))
prefer *2* **apply** (*simp add*: *Un-upper2 contin-imp-mono*)
apply (*simp add*: *contin-def*, *clarify*)
apply (*rule equalityI*)
prefer *2* **apply** *blast*
apply *clarify*
apply (*rename-tac B C*)
apply (*rule-tac a*=*B* ∪ *C* **in** *UN-I*)
apply (*simp add*: *directed-def*, *blast*)
done

lemma *const-contin*: *contin*(λ*X*. *A*)
by (*simp add*: *contin-def directed-def*)

lemma *id-contin*: *contin*(λ*X*. *X*)
by (*simp add*: *contin-def*)

6.2 Absoluteness for "Iterates"

definition
  iterates-MH :: [*i*=>*o*, [*i*,*i*]=>*o*, *i*, *i*, *i*, *i*] => *o* **where**
   iterates-MH(*M*,*isF*,*v*,*n*,*g*,*z*) ==
        is-nat-case(*M*, *v*, λ*m u*. ∃ *gm*[*M*]. *fun-apply*(*M*,*g*,*m*,*gm*) & *isF*(*gm*,*u*),
                n, *z*)

definition
  is-iterates :: [*i*=>*o*, [*i*,*i*]=>*o*, *i*, *i*, *i*] => *o* **where**
    is-iterates(*M*,*isF*,*v*,*n*,*Z*) ==
      ∃ *sn*[*M*]. ∃ *msn*[*M*]. *successor*(*M*,*n*,*sn*) & *membership*(*M*,*sn*,*msn*) &
                is-wfrec(*M*, *iterates-MH*(*M*,*isF*,*v*), *msn*, *n*, *Z*)

definition
  iterates-replacement :: [*i*=>*o*, [*i*,*i*]=>*o*, *i*] => *o* **where**
   iterates-replacement(*M*,*isF*,*v*) ==
      ∀ *n*[*M*]. *n*∈*nat* ⟶
         wfrec-replacement(*M*, *iterates-MH*(*M*,*isF*,*v*), *Memrel*(*succ*(*n*)))

lemma (**in** *M-basic*) *iterates-MH-abs*:
  [| *relation1*(*M*,*isF*,*F*); *M*(*n*); *M*(*g*); *M*(*z*) |]
   ==> *iterates-MH*(*M*,*isF*,*v*,*n*,*g*,*z*) ⟷ *z* = *nat-case*(*v*, λ*m*. *F*(*g*‘*m*), *n*)
by (*simp add*: *nat-case-abs* [*of* - λ*m*. *F*(*g* ‘ *m*)]
            relation1-def iterates-MH-def)

lemma (**in** *M-trancl*) *iterates-imp-wfrec-replacement*:
  [|*relation1*(*M*,*isF*,*F*); *n* ∈ *nat*; *iterates-replacement*(*M*,*isF*,*v*)|]
   ==> *wfrec-replacement*(*M*, λ*n f z*. *z* = *nat-case*(*v*, λ*m*. *F*(*f*‘*m*), *n*),
                Memrel(*succ*(*n*)))
by (*simp add*: *iterates-replacement-def iterates-MH-abs*)

theorem (**in** *M-trancl*) *iterates-abs*:
  [| *iterates-replacement*(*M*,*isF*,*v*); *relation1*(*M*,*isF*,*F*);
      n ∈ *nat*; *M*(*v*); *M*(*z*); ∀ *x*[*M*]. *M*(*F*(*x*)) |]
   ==> *is-iterates*(*M*,*isF*,*v*,*n*,*z*) ⟷ *z* = *iterates*(*F*,*n*,*v*)
apply (*frule iterates-imp-wfrec-replacement*, *assumption*+)
apply (*simp add*: *wf-Memrel trans-Memrel relation-Memrel*
                is-iterates-def relation2-def iterates-MH-abs
                iterates-nat-def recursor-def transrec-def
                eclose-sing-Ord-eq nat-into-M
         trans-wfrec-abs [*of* - - - - λ*n g*. *nat-case*(*v*, λ*m*. *F*(*g*‘*m*), *n*)])
done

lemma (**in** *M-trancl*) *iterates-closed* [*intro*,*simp*]:
  [| *iterates-replacement*(*M*,*isF*,*v*); *relation1*(*M*,*isF*,*F*);
      n ∈ *nat*; *M*(*v*); ∀ *x*[*M*]. *M*(*F*(*x*)) |]
   ==> *M*(*iterates*(*F*,*n*,*v*))
apply (*frule iterates-imp-wfrec-replacement*, *assumption*+)

apply (*simp add*: *wf-Memrel trans-Memrel relation-Memrel*
relation2-def iterates-MH-abs
iterates-nat-def recursor-def transrec-def
eclose-sing-Ord-eq nat-into-M
trans-wfrec-closed [*of - - -* $\lambda n\ g.\ nat\text{-}case(v,\ \lambda m.\ F(g`m),\ n)$])
done

6.3 lists without univ

lemmas *datatype-univs* = *Inl-in-univ Inr-in-univ*
Pair-in-univ nat-into-univ A-into-univ

lemma *list-fun-bnd-mono*: $bnd\text{-}mono(univ(A),\ \lambda X.\ \{0\} + A*X)$
apply (*rule bnd-monoI*)
apply (*intro subset-refl zero-subset-univ A-subset-univ*
sum-subset-univ Sigma-subset-univ)
apply (*rule subset-refl sum-mono Sigma-mono* | *assumption*)+
done

lemma *list-fun-contin*: $contin(\lambda X.\ \{0\} + A*X)$
by (*intro sum-contin prod-contin id-contin const-contin*)

Re-expresses lists using sum and product

lemma *list-eq-lfp2*: $list(A) = lfp(univ(A),\ \lambda X.\ \{0\} + A*X)$
apply (*simp add*: *list-def*)
apply (*rule equalityI*)
apply (*rule lfp-lowerbound*)
prefer *2* **apply** (*rule lfp-subset*)
apply (*clarify*, *subst lfp-unfold* [*OF list-fun-bnd-mono*])
apply (*simp add*: *Nil-def Cons-def*)
apply *blast*

Opposite inclusion

apply (*rule lfp-lowerbound*)
prefer *2* **apply** (*rule lfp-subset*)
apply (*clarify*, *subst lfp-unfold* [*OF list.bnd-mono*])
apply (*simp add*: *Nil-def Cons-def*)
apply (*blast intro*: *datatype-univs*
dest: *lfp-subset* [*THEN subsetD*])
done

Re-expresses lists using "iterates", no univ.

lemma *list-eq-Union*:
$list(A) = (\bigcup n\in nat.\ (\lambda X.\ \{0\} + A*X)\ \hat{}\ n\ (0))$
by (*simp add*: *list-eq-lfp2 lfp-eq-Union list-fun-bnd-mono list-fun-contin*)

definition
is-list-functor :: [*i=>o,i,i,i*] => *o* **where**

is-list-functor(*M*,*A*,*X*,*Z*) ==
∃ *n1*[*M*]. ∃ *AX*[*M*].
number1(*M*,*n1*) & *cartprod*(*M*,*A*,*X*,*AX*) & *is-sum*(*M*,*n1*,*AX*,*Z*)

lemma (**in** *M-basic*) *list-functor-abs* [*simp*]:
[| *M*(*A*); *M*(*X*); *M*(*Z*) |] ==> *is-list-functor*(*M*,*A*,*X*,*Z*) ⟷ (*Z* = {*0*} + *A*∗*X*)
by (*simp add*: *is-list-functor-def singleton-0 nat-into-M*)

6.4 formulas without univ

lemma *formula-fun-bnd-mono*:
bnd-mono(*univ*(*0*), λ*X*. ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*X*∗*X* + *X*))
apply (*rule bnd-monoI*)
apply (*intro subset-refl zero-subset-univ A-subset-univ sum-subset-univ Sigma-subset-univ nat-subset-univ*)
apply (*rule subset-refl sum-mono Sigma-mono* | *assumption*)+
done

lemma *formula-fun-contin*:
contin(λ*X*. ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*X*∗*X* + *X*))
by (*intro sum-contin prod-contin id-contin const-contin*)

Re-expresses formulas using sum and product

lemma *formula-eq-lfp2*:
formula = *lfp*(*univ*(*0*), λ*X*. ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*X*∗*X* + *X*))
apply (*simp add*: *formula-def*)
apply (*rule equalityI*)
apply (*rule lfp-lowerbound*)
prefer *2* **apply** (*rule lfp-subset*)
apply (*clarify*, *subst lfp-unfold* [*OF formula-fun-bnd-mono*])
apply (*simp add*: *Member-def Equal-def Nand-def Forall-def*)
apply *blast*

Opposite inclusion

apply (*rule lfp-lowerbound*)
prefer *2* **apply** (*rule lfp-subset*, *clarify*)
apply (*subst lfp-unfold* [*OF formula.bnd-mono*, *simplified*])
apply (*simp add*: *Member-def Equal-def Nand-def Forall-def*)
apply (*elim sumE SigmaE*, *simp-all*)
apply (*blast intro*: *datatype-univs dest*: *lfp-subset* [*THEN subsetD*])+
done

Re-expresses formulas using "iterates", no univ.

lemma *formula-eq-Union*:
formula =
(⋃*n*∈*nat*. (λ*X*. ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*X*∗*X* + *X*)) ^ *n* (*0*))
by (*simp add*: *formula-eq-lfp2 lfp-eq-Union formula-fun-bnd-mono formula-fun-contin*)

definition
is-formula-functor :: [*i*=>*o*,*i*,*i*] => *o* **where**
is-formula-functor(*M*,*X*,*Z*) ==
∃ *nat'*[*M*]. ∃ *natnat*[*M*]. ∃ *natnatsum*[*M*]. ∃ *XX*[*M*]. ∃ *X3*[*M*].
omega(*M*,*nat'*) & *cartprod*(*M*,*nat'*,*nat'*,*natnat*) &
is-sum(*M*,*natnat*,*natnat*,*natnatsum*) &
cartprod(*M*,*X*,*X*,*XX*) & *is-sum*(*M*,*XX*,*X*,*X3*) &
is-sum(*M*,*natnatsum*,*X3*,*Z*)

lemma (**in** *M-trancl*) *formula-functor-abs* [*simp*]:
[| *M*(*X*); *M*(*Z*) |]
==> *is-formula-functor*(*M*,*X*,*Z*) ⟷
Z = ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*X*∗*X* + *X*)
by (*simp add*: *is-formula-functor-def*)

6.5 *M* Contains the List and Formula Datatypes

definition
list-N :: [*i*,*i*] => *i* **where**
list-N(*A*,*n*) == (λ*X*. {*0*} + *A* ∗ *X*) ^*n* (*0*)

lemma *Nil-in-list-N* [*simp*]: [] ∈ *list-N*(*A*,*succ*(*n*))
by (*simp add*: *list-N-def Nil-def*)

lemma *Cons-in-list-N* [*simp*]:
Cons(*a*,*l*) ∈ *list-N*(*A*,*succ*(*n*)) ⟷ *a*∈*A* & *l* ∈ *list-N*(*A*,*n*)
by (*simp add*: *list-N-def Cons-def*)

These two aren't simprules because they reveal the underlying list representation.

lemma *list-N-0*: *list-N*(*A*,*0*) = *0*
by (*simp add*: *list-N-def*)

lemma *list-N-succ*: *list-N*(*A*,*succ*(*n*)) = {*0*} + *A* ∗ (*list-N*(*A*,*n*))
by (*simp add*: *list-N-def*)

lemma *list-N-imp-list*:
[| *l* ∈ *list-N*(*A*,*n*); *n* ∈ *nat* |] ==> *l* ∈ *list*(*A*)
by (*force simp add*: *list-eq-Union list-N-def*)

lemma *list-N-imp-length-lt* [*rule-format*]:
n ∈ *nat* ==> ∀ *l* ∈ *list-N*(*A*,*n*). *length*(*l*) < *n*
apply (*induct-tac n*)
apply (*auto simp add*: *list-N-0 list-N-succ*
Nil-def [*symmetric*] *Cons-def* [*symmetric*])
done

lemma *list-imp-list-N* [*rule-format*]:
$l \in list(A) \Longrightarrow \forall n \in nat.\ length(l) < n \longrightarrow l \in list\text{-}N(A,\ n)$
apply (*induct-tac l*)
apply (*force elim*: *natE*)+
done

lemma *list-N-imp-eq-length*:
$[|n \in nat;\ l \notin list\text{-}N(A,\ n);\ l \in list\text{-}N(A,\ succ(n))|]$
$\Longrightarrow n = length(l)$
apply (*rule le-anti-sym*)
prefer *2* **apply** (*simp add*: *list-N-imp-length-lt*)
apply (*frule list-N-imp-list*, *simp*)
apply (*simp add*: *not-lt-iff-le* [*symmetric*])
apply (*blast intro*: *list-imp-list-N*)
done

Express *list-rec* without using *rank* or *Vset*, neither of which is absolute.

lemma (**in** *M-trivial*) *list-rec-eq*:
$l \in list(A) \Longrightarrow$
$list\text{-}rec(a,g,l) =$
$transrec\ (succ(length(l)),$
$\lambda x\ h.\ Lambda\ (list(A),$
$list\text{-}case'\ (a,$
$\lambda a\ l.\ g(a,\ l,\ h\ `\ succ(length(l))\ `\ l)))) \ `\ l$
apply (*induct-tac l*)
apply (*subst transrec*, *simp*)
apply (*subst transrec*)
apply (*simp add*: *list-imp-list-N*)
done

definition
is-list-N :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
$is\text{-}list\text{-}N(M,A,n,Z) ==$
$\exists zero[M].\ empty(M,zero)\ \&$
$is\text{-}iterates(M,\ is\text{-}list\text{-}functor(M,A),\ zero,\ n,\ Z)$

definition
mem-list :: [*i*=>*o*,*i*,*i*] => *o* **where**
$mem\text{-}list(M,A,l) ==$
$\exists n[M].\ \exists listn[M].$
$finite\text{-}ordinal(M,n)\ \&\ is\text{-}list\text{-}N(M,A,n,listn)\ \&\ l \in listn$

definition
is-list :: [*i*=>*o*,*i*,*i*] => *o* **where**
$is\text{-}list(M,A,Z) == \forall l[M].\ l \in Z \longleftrightarrow mem\text{-}list(M,A,l)$

6.5.1 Towards Absoluteness of *formula-rec*

consts *depth* :: *i*=>*i*

primrec
  depth(*Member*(*x*,*y*)) = *0*
  depth(*Equal*(*x*,*y*)) = *0*
  depth(*Nand*(*p*,*q*)) = *succ*(*depth*(*p*) ∪ *depth*(*q*))
  depth(*Forall*(*p*)) = *succ*(*depth*(*p*))

lemma *depth-type* [*TC*]: *p* ∈ *formula* ==> *depth*(*p*) ∈ *nat*
by (*induct-tac p*, *simp-all*)

definition
  formula-N :: *i* => *i* **where**
    formula-N(*n*) == (λ*X*. ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*X*∗*X* + *X*)) ^ *n* (*0*)

lemma *Member-in-formula-N* [*simp*]:
     Member(*x*,*y*) ∈ *formula-N*(*succ*(*n*)) ⟷ *x* ∈ *nat* & *y* ∈ *nat*
by (*simp add*: *formula-N-def Member-def*)

lemma *Equal-in-formula-N* [*simp*]:
     Equal(*x*,*y*) ∈ *formula-N*(*succ*(*n*)) ⟷ *x* ∈ *nat* & *y* ∈ *nat*
by (*simp add*: *formula-N-def Equal-def*)

lemma *Nand-in-formula-N* [*simp*]:
     Nand(*x*,*y*) ∈ *formula-N*(*succ*(*n*)) ⟷ *x* ∈ *formula-N*(*n*) & *y* ∈ *formula-N*(*n*)
by (*simp add*: *formula-N-def Nand-def*)

lemma *Forall-in-formula-N* [*simp*]:
     Forall(*x*) ∈ *formula-N*(*succ*(*n*)) ⟷ *x* ∈ *formula-N*(*n*)
by (*simp add*: *formula-N-def Forall-def*)

These two aren't simprules because they reveal the underlying formula representation.

lemma *formula-N-0*: *formula-N*(*0*) = *0*
by (*simp add*: *formula-N-def*)

lemma *formula-N-succ*:
     formula-N(*succ*(*n*)) =
      ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*formula-N*(*n*) ∗ *formula-N*(*n*) + *formula-N*(*n*))
by (*simp add*: *formula-N-def*)

lemma *formula-N-imp-formula*:
  [| *p* ∈ *formula-N*(*n*); *n* ∈ *nat* |] ==> *p* ∈ *formula*
by (*force simp add*: *formula-eq-Union formula-N-def*)

lemma *formula-N-imp-depth-lt* [*rule-format*]:
     n ∈ *nat* ==> ∀ *p* ∈ *formula-N*(*n*). *depth*(*p*) < *n*
apply (*induct-tac n*)
apply (*auto simp add*: *formula-N-0 formula-N-succ*
                    depth-type formula-N-imp-formula Un-least-lt-iff

Member-def [*symmetric*] *Equal-def* [*symmetric*]
Nand-def [*symmetric*] *Forall-def* [*symmetric*])
done

lemma *formula-imp-formula-N* [*rule-format*]:
$p \in formula ==> \forall n{\in}nat.\ depth(p) < n \longrightarrow p \in formula\text{-}N(n)$
apply (*induct-tac p*)
apply (*simp-all add*: *succ-Un-distrib Un-least-lt-iff*)
apply (*force elim*: *natE*)+
done

lemma *formula-N-imp-eq-depth*:
$[|n \in nat;\ p \notin formula\text{-}N(n);\ p \in formula\text{-}N(succ(n))|]$
$==> n = depth(p)$
apply (*rule le-anti-sym*)
prefer *2* **apply** (*simp add*: *formula-N-imp-depth-lt*)
apply (*frule formula-N-imp-formula*, *simp*)
apply (*simp add*: *not-lt-iff-le* [*symmetric*])
apply (*blast intro*: *formula-imp-formula-N*)
done

This result and the next are unused.

lemma *formula-N-mono* [*rule-format*]:
$[|\ m \in nat;\ n \in nat\ |] ==> m{\le}n \longrightarrow formula\text{-}N(m) \subseteq formula\text{-}N(n)$
apply (*rule-tac m = m* **and** *n = n* **in** *diff-induct*)
apply (*simp-all add*: *formula-N-0 formula-N-succ*, *blast*)
done

lemma *formula-N-distrib*:
$[|\ m \in nat;\ n \in nat\ |] ==> formula\text{-}N(m \cup n) = formula\text{-}N(m) \cup formula\text{-}N(n)$
apply (*rule-tac i = m* **and** *j = n* **in** *Ord-linear-le*, *auto*)
apply (*simp-all add*: *subset-Un-iff* [*THEN iffD1*] *subset-Un-iff2* [*THEN iffD1*]
le-imp-subset formula-N-mono)
done

definition
is-formula-N :: [*i*=>*o*,*i*,*i*] => *o* **where**
is-formula-N(*M*,*n*,*Z*) ==
∃ *zero*[*M*]. *empty*(*M*,*zero*) &
is-iterates(*M*, *is-formula-functor*(*M*), *zero*, *n*, *Z*)

definition
mem-formula :: [*i*=>*o*,*i*] => *o* **where**
mem-formula(*M*,*p*) ==
∃ *n*[*M*]. ∃ *formn*[*M*].
finite-ordinal(*M*,*n*) & *is-formula-N*(*M*,*n*,*formn*) & *p* ∈ *formn*

definition

is-formula :: [*i*=>*o*,*i*] => *o* **where**
is-formula(*M*,*Z*) == ∀ *p*[*M*]. *p* ∈ *Z* ⟷ *mem-formula*(*M*,*p*)

locale *M-datatypes* = *M-trancl* +
assumes *list-replacement1*:
M(*A*) ==> *iterates-replacement*(*M*, *is-list-functor*(*M*,*A*), *0*)
and *list-replacement2*:
M(*A*) ==> *strong-replacement*(*M*,
λ*n y*. *n*∈*nat* & *is-iterates*(*M*, *is-list-functor*(*M*,*A*), *0*, *n*, *y*))
and *formula-replacement1*:
iterates-replacement(*M*, *is-formula-functor*(*M*), *0*)
and *formula-replacement2*:
strong-replacement(*M*,
λ*n y*. *n*∈*nat* & *is-iterates*(*M*, *is-formula-functor*(*M*), *0*, *n*, *y*))
and *nth-replacement*:
M(*l*) ==> *iterates-replacement*(*M*, %*l t*. *is-tl*(*M*,*l*,*t*), *l*)

6.5.2 Absoluteness of the List Construction

lemma (**in** *M-datatypes*) *list-replacement2'*:
M(*A*) ==> *strong-replacement*(*M*, λ*n y*. *n*∈*nat* & *y* = (λ*X*. {*0*} + *A* ∗ *X*) ^*n* (*0*))
apply (*insert list-replacement2* [*of A*])
apply (*rule strong-replacement-cong* [*THEN iffD1*])
apply (*rule conj-cong* [*OF iff-refl iterates-abs* [*of is-list-functor*(*M*,*A*)]])
apply (*simp-all add*: *list-replacement1 relation1-def*)
done

lemma (**in** *M-datatypes*) *list-closed* [*intro*,*simp*]:
M(*A*) ==> *M*(*list*(*A*))
apply (*insert list-replacement1*)
by (*simp add*: *RepFun-closed2 list-eq-Union*
list-replacement2' relation1-def
iterates-closed [*of is-list-functor*(*M*,*A*)])

WARNING: use only with *dest*: or with variables fixed!

lemmas (**in** *M-datatypes*) *list-into-M* = *transM* [*OF* - *list-closed*]

lemma (**in** *M-datatypes*) *list-N-abs* [*simp*]:
[|*M*(*A*); *n*∈*nat*; *M*(*Z*)|]
==> *is-list-N*(*M*,*A*,*n*,*Z*) ⟷ *Z* = *list-N*(*A*,*n*)
apply (*insert list-replacement1*)
apply (*simp add*: *is-list-N-def list-N-def relation1-def nat-into-M*
iterates-abs [*of is-list-functor*(*M*,*A*) - λ*X*. {*0*} + *A*∗*X*])
done

lemma (**in** *M-datatypes*) *list-N-closed* [*intro*,*simp*]:
[|*M*(*A*); *n*∈*nat*|] ==> *M*(*list-N*(*A*,*n*))
apply (*insert list-replacement1*)

apply (*simp add*: *is-list-N-def list-N-def relation1-def nat-into-M*
iterates-closed [*of is-list-functor*(*M*,*A*)])
done

lemma (**in** *M-datatypes*) *mem-list-abs* [*simp*]:
M(*A*) ==> *mem-list*(*M*,*A*,*l*) ⟷ *l* ∈ *list*(*A*)
apply (*insert list-replacement1*)
apply (*simp add*: *mem-list-def list-N-def relation1-def list-eq-Union*
iterates-closed [*of is-list-functor*(*M*,*A*)])
done

lemma (**in** *M-datatypes*) *list-abs* [*simp*]:
[|*M*(*A*); *M*(*Z*)|] ==> *is-list*(*M*,*A*,*Z*) ⟷ *Z* = *list*(*A*)
apply (*simp add*: *is-list-def*, *safe*)
apply (*rule M-equalityI*, *simp-all*)
done

6.5.3 Absoluteness of Formulas

lemma (**in** *M-datatypes*) *formula-replacement2'*:
strong-replacement(*M*, λ*n y*. *n*∈*nat* & *y* = (λ*X*. ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*X*∗*X* + *X*))ˆ*n* (*0*))
apply (*insert formula-replacement2*)
apply (*rule strong-replacement-cong* [*THEN iffD1*])
apply (*rule conj-cong* [*OF iff-refl iterates-abs* [*of is-formula-functor*(*M*)]])
apply (*simp-all add*: *formula-replacement1 relation1-def*)
done

lemma (**in** *M-datatypes*) *formula-closed* [*intro*,*simp*]:
M(*formula*)
apply (*insert formula-replacement1*)
apply (*simp add*: *RepFun-closed2 formula-eq-Union*
formula-replacement2' relation1-def
iterates-closed [*of is-formula-functor*(*M*)])
done

lemmas (**in** *M-datatypes*) *formula-into-M* = *transM* [*OF - formula-closed*]

lemma (**in** *M-datatypes*) *formula-N-abs* [*simp*]:
[|*n*∈*nat*; *M*(*Z*)|]
==> *is-formula-N*(*M*,*n*,*Z*) ⟷ *Z* = *formula-N*(*n*)
apply (*insert formula-replacement1*)
apply (*simp add*: *is-formula-N-def formula-N-def relation1-def nat-into-M*
iterates-abs [*of is-formula-functor*(*M*) -
λ*X*. ((*nat*∗*nat*) + (*nat*∗*nat*)) + (*X*∗*X* + *X*)])
done

lemma (**in** *M-datatypes*) *formula-N-closed* [*intro*,*simp*]:
n∈*nat* ==> *M*(*formula-N*(*n*))

apply (*insert formula-replacement1*)
apply (*simp add*: *is-formula-N-def formula-N-def relation1-def nat-into-M*
iterates-closed [*of is-formula-functor*(*M*)])
done

lemma (**in** *M-datatypes*) *mem-formula-abs* [*simp*]:
mem-formula(*M*,*l*) ⟷ *l* ∈ *formula*
apply (*insert formula-replacement1*)
apply (*simp add*: *mem-formula-def relation1-def formula-eq-Union formula-N-def*
iterates-closed [*of is-formula-functor*(*M*)])
done

lemma (**in** *M-datatypes*) *formula-abs* [*simp*]:
[|*M*(*Z*)|] ==> *is-formula*(*M*,*Z*) ⟷ *Z* = *formula*
apply (*simp add*: *is-formula-def*, *safe*)
apply (*rule M-equalityI*, *simp-all*)
done

6.6 Absoluteness for ε-Closure: the *eclose* Operator

Re-expresses eclose using "iterates"

lemma *eclose-eq-Union*:
eclose(*A*) = (⋃*n*∈*nat*. *Union*^*n* (*A*))
apply (*simp add*: *eclose-def*)
apply (*rule UN-cong*)
apply (*rule refl*)
apply (*induct-tac n*)
apply (*simp add*: *nat-rec-0*)
apply (*simp add*: *nat-rec-succ*)
done

definition
is-eclose-n :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
is-eclose-n(*M*,*A*,*n*,*Z*) == *is-iterates*(*M*, *big-union*(*M*), *A*, *n*, *Z*)

definition
mem-eclose :: [*i*=>*o*,*i*,*i*] => *o* **where**
mem-eclose(*M*,*A*,*l*) ==
∃ *n*[*M*]. ∃ *eclosen*[*M*].
finite-ordinal(*M*,*n*) & *is-eclose-n*(*M*,*A*,*n*,*eclosen*) & *l* ∈ *eclosen*

definition
is-eclose :: [*i*=>*o*,*i*,*i*] => *o* **where**
is-eclose(*M*,*A*,*Z*) == ∀ *u*[*M*]. *u* ∈ *Z* ⟷ *mem-eclose*(*M*,*A*,*u*)

locale *M-eclose* = *M-datatypes* +
assumes *eclose-replacement1*:
M(*A*) ==> *iterates-replacement*(*M*, *big-union*(*M*), *A*)

and *eclose-replacement2*:
M(*A*) ==> *strong-replacement*(*M*,
λn y. *n*∈*nat* & *is-iterates*(*M*, *big-union*(*M*), *A*, *n*, *y*))

lemma (**in** *M-eclose*) *eclose-replacement2'*:
M(*A*) ==> *strong-replacement*(*M*, *λn y*. *n*∈*nat* & *y* = *Union*ˆ*n* (*A*))
apply (*insert eclose-replacement2* [*of A*])
apply (*rule strong-replacement-cong* [*THEN iffD1*])
apply (*rule conj-cong* [*OF iff-refl iterates-abs* [*of big-union*(*M*)]])
apply (*simp-all add*: *eclose-replacement1 relation1-def*)
done

lemma (**in** *M-eclose*) *eclose-closed* [*intro*,*simp*]:
M(*A*) ==> *M*(*eclose*(*A*))
apply (*insert eclose-replacement1*)
by (*simp add*: *RepFun-closed2 eclose-eq-Union*
eclose-replacement2' relation1-def
iterates-closed [*of big-union*(*M*)])

lemma (**in** *M-eclose*) *is-eclose-n-abs* [*simp*]:
[|*M*(*A*); *n*∈*nat*; *M*(*Z*)|] ==> *is-eclose-n*(*M*,*A*,*n*,*Z*) ⟷ *Z* = *Union*ˆ*n* (*A*)
apply (*insert eclose-replacement1*)
apply (*simp add*: *is-eclose-n-def relation1-def nat-into-M*
iterates-abs [*of big-union*(*M*) - *Union*])
done

lemma (**in** *M-eclose*) *mem-eclose-abs* [*simp*]:
M(*A*) ==> *mem-eclose*(*M*,*A*,*l*) ⟷ *l* ∈ *eclose*(*A*)
apply (*insert eclose-replacement1*)
apply (*simp add*: *mem-eclose-def relation1-def eclose-eq-Union*
iterates-closed [*of big-union*(*M*)])
done

lemma (**in** *M-eclose*) *eclose-abs* [*simp*]:
[|*M*(*A*); *M*(*Z*)|] ==> *is-eclose*(*M*,*A*,*Z*) ⟷ *Z* = *eclose*(*A*)
apply (*simp add*: *is-eclose-def*, *safe*)
apply (*rule M-equalityI*, *simp-all*)
done

6.7 Absoluteness for *transrec*

transrec(*a*, *H*) ≡ *wfrec*(*Memrel*(*eclose*({*a*})), *a*, *H*)

definition
is-transrec :: [*i*=>*o*, [*i*,*i*,*i*]=>*o*, *i*, *i*] => *o* **where**
is-transrec(*M*,*MH*,*a*,*z*) ==
∃ *sa*[*M*]. ∃ *esa*[*M*]. ∃ *mesa*[*M*].
upair(*M*,*a*,*a*,*sa*) & *is-eclose*(*M*,*sa*,*esa*) & *membership*(*M*,*esa*,*mesa*) &
is-wfrec(*M*,*MH*,*mesa*,*a*,*z*)

definition
transrec-replacement :: [i=>o, [i,i,i]=>o, i] => o **where**
transrec-replacement(M,MH,a) ==
∃ sa[M]. ∃ esa[M]. ∃ mesa[M].
upair(M,a,a,sa) & *is-eclose*(M,sa,esa) & *membership*(M,esa,mesa) &
wfrec-replacement(M,MH,mesa)

The condition *Ord*(i) lets us use the simpler *trans-wfrec-abs* rather than *trans-wfrec-abs*, which I haven't even proved yet.

theorem (**in** *M-eclose*) *transrec-abs*:
[|*transrec-replacement*(M,MH,i); *relation2*(M,MH,H);
Ord(i); M(i); M(z);
∀ x[M]. ∀ g[M]. *function*(g) ⟶ M(H(x,g))|]
==> *is-transrec*(M,MH,i,z) ⟷ z = *transrec*(i,H)
by (*simp add*: *trans-wfrec-abs transrec-replacement-def is-transrec-def*
transrec-def eclose-sing-Ord-eq wf-Memrel trans-Memrel relation-Memrel)

theorem (**in** *M-eclose*) *transrec-closed*:
[|*transrec-replacement*(M,MH,i); *relation2*(M,MH,H);
Ord(i); M(i);
∀ x[M]. ∀ g[M]. *function*(g) ⟶ M(H(x,g))|]
==> M(*transrec*(i,H))
by (*simp add*: *trans-wfrec-closed transrec-replacement-def is-transrec-def*
transrec-def eclose-sing-Ord-eq wf-Memrel trans-Memrel relation-Memrel)

Helps to prove instances of *transrec-replacement*

lemma (**in** *M-eclose*) *transrec-replacementI*:
[|M(a);
strong-replacement (M,
λx z. ∃ y[M]. *pair*(M, x, y, z) &
is-wfrec(M,MH,*Memrel*(*eclose*({a})),x,y))|]
==> *transrec-replacement*(M,MH,a)
by (*simp add*: *transrec-replacement-def wfrec-replacement-def*)

6.8 Absoluteness for the List Operator *length*

But it is never used.

definition
is-length :: [i=>o,i,i,i] => o **where**
is-length(M,A,l,n) ==
∃ sn[M]. ∃ *list-n*[M]. ∃ *list-sn*[M].
is-list-N(M,A,n,*list-n*) & l ∉ *list-n* &
successor(M,n,sn) & *is-list-N*(M,A,sn,*list-sn*) & l ∈ *list-sn*

lemma (**in** *M-datatypes*) *length-abs* [*simp*]:
[|M(A); l ∈ *list*(A); n ∈ *nat*|] ==> *is-length*(M,A,l,n) ⟷ n = *length*(l)

apply (*subgoal-tac* $M(l)$ & $M(n)$)
prefer *2* **apply** (*blast dest*: *transM*)
apply (*simp add*: *is-length-def*)
apply (*blast intro*: *list-imp-list-N nat-into-Ord list-N-imp-eq-length*
dest: *list-N-imp-length-lt*)
done

Proof is trivial since *length* returns natural numbers.

lemma (**in** *M-trivial*) *length-closed* [*intro*,*simp*]:
$l \in list(A) ==> M(length(l))$
by (*simp add*: *nat-into-M*)

6.9 Absoluteness for the List Operator *nth*

lemma *nth-eq-hd-iterates-tl* [*rule-format*]:
$xs \in list(A) ==> \forall n \in nat.\ nth(n,xs) = hd'\ (tl'\hat{}n\ (xs))$
apply (*induct-tac xs*)
apply (*simp add*: *iterates-tl-Nil hd'-Nil*, *clarify*)
apply (*erule natE*)
apply (*simp add*: *hd'-Cons*)
apply (*simp add*: *tl'-Cons iterates-commute*)
done

lemma (**in** *M-basic*) *iterates-tl'-closed*:
$[|n \in nat;\ M(x)|] ==> M(tl'\hat{}n\ (x))$
apply (*induct-tac n*, *simp*)
apply (*simp add*: *tl'-Cons tl'-closed*)
done

Immediate by type-checking

lemma (**in** *M-datatypes*) *nth-closed* [*intro*,*simp*]:
$[|xs \in list(A);\ n \in nat;\ M(A)|] ==> M(nth(n,xs))$
apply (*case-tac* $n < length(xs)$)
apply (*blast intro*: *nth-type transM*)
apply (*simp add*: *not-lt-iff-le nth-eq-0*)
done

definition
is-nth :: $[i=>o,i,i,i] => o$ **where**
$is\text{-}nth(M,n,l,Z) ==$
$\exists X[M].\ is\text{-}iterates(M,\ is\text{-}tl(M),\ l,\ n,\ X)\ \&\ is\text{-}hd(M,X,Z)$

lemma (**in** *M-datatypes*) *nth-abs* [*simp*]:
$[|M(A);\ n \in nat;\ l \in list(A);\ M(Z)|]$
$==> is\text{-}nth(M,n,l,Z) \longleftrightarrow Z = nth(n,l)$
apply (*subgoal-tac* $M(l)$)
prefer *2* **apply** (*blast intro*: *transM*)
apply (*simp add*: *is-nth-def nth-eq-hd-iterates-tl nat-into-M*
tl'-closed iterates-tl'-closed

iterates-abs [*OF* - *relation1-tl*] *nth-replacement*)
done

6.10 Relativization and Absoluteness for the *formula* Constructors

definition
is-Member :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
— because *Member*(*x*, *y*) ≡ *Inl*(*Inl*(⟨*x*, *y*⟩))
is-Member(*M*,*x*,*y*,*Z*) ==
∃ *p*[*M*]. ∃ *u*[*M*]. *pair*(*M*,*x*,*y*,*p*) & *is-Inl*(*M*,*p*,*u*) & *is-Inl*(*M*,*u*,*Z*)

lemma (**in** *M-trivial*) *Member-abs* [*simp*]:
[|*M*(*x*); *M*(*y*); *M*(*Z*)|] ==> *is-Member*(*M*,*x*,*y*,*Z*) ⟷ (*Z* = *Member*(*x*,*y*))
by (*simp add*: *is-Member-def Member-def*)

lemma (**in** *M-trivial*) *Member-in-M-iff* [*iff*]:
M(*Member*(*x*,*y*)) ⟷ *M*(*x*) & *M*(*y*)
by (*simp add*: *Member-def*)

definition
is-Equal :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
— because *Equal*(*x*, *y*) ≡ *Inl*(*Inr*(⟨*x*, *y*⟩))
is-Equal(*M*,*x*,*y*,*Z*) ==
∃ *p*[*M*]. ∃ *u*[*M*]. *pair*(*M*,*x*,*y*,*p*) & *is-Inr*(*M*,*p*,*u*) & *is-Inl*(*M*,*u*,*Z*)

lemma (**in** *M-trivial*) *Equal-abs* [*simp*]:
[|*M*(*x*); *M*(*y*); *M*(*Z*)|] ==> *is-Equal*(*M*,*x*,*y*,*Z*) ⟷ (*Z* = *Equal*(*x*,*y*))
by (*simp add*: *is-Equal-def Equal-def*)

lemma (**in** *M-trivial*) *Equal-in-M-iff* [*iff*]: *M*(*Equal*(*x*,*y*)) ⟷ *M*(*x*) & *M*(*y*)
by (*simp add*: *Equal-def*)

definition
is-Nand :: [*i*=>*o*,*i*,*i*,*i*] => *o* **where**
— because *Nand*(*x*, *y*) ≡ *Inr*(*Inl*(⟨*x*, *y*⟩))
is-Nand(*M*,*x*,*y*,*Z*) ==
∃ *p*[*M*]. ∃ *u*[*M*]. *pair*(*M*,*x*,*y*,*p*) & *is-Inl*(*M*,*p*,*u*) & *is-Inr*(*M*,*u*,*Z*)

lemma (**in** *M-trivial*) *Nand-abs* [*simp*]:
[|*M*(*x*); *M*(*y*); *M*(*Z*)|] ==> *is-Nand*(*M*,*x*,*y*,*Z*) ⟷ (*Z* = *Nand*(*x*,*y*))
by (*simp add*: *is-Nand-def Nand-def*)

lemma (**in** *M-trivial*) *Nand-in-M-iff* [*iff*]: *M*(*Nand*(*x*,*y*)) ⟷ *M*(*x*) & *M*(*y*)
by (*simp add*: *Nand-def*)

definition
is-Forall :: [*i*=>*o*,*i*,*i*] => *o* **where**
— because *Forall*(*x*) ≡ *Inr*(*Inr*(*p*))

is-Forall(*M*,*p*,*Z*) == ∃ *u*[*M*]. *is-Inr*(*M*,*p*,*u*) & *is-Inr*(*M*,*u*,*Z*)

lemma (**in** *M-trivial*) *Forall-abs* [*simp*]:
[|*M*(*x*); *M*(*Z*)|] ==> *is-Forall*(*M*,*x*,*Z*) ⟷ (*Z* = *Forall*(*x*))
by (*simp add*: *is-Forall-def Forall-def*)

lemma (**in** *M-trivial*) *Forall-in-M-iff* [*iff*]: *M*(*Forall*(*x*)) ⟷ *M*(*x*)
by (*simp add*: *Forall-def*)

6.11 Absoluteness for *formula-rec*

definition
formula-rec-case :: [[*i*,*i*]=>*i*, [*i*,*i*]=>*i*, [*i*,*i*,*i*,*i*]=>*i*, [*i*,*i*]=>*i*, *i*, *i*] => *i* **where**
— the instance of *formula-case* in *formula-rec*
formula-rec-case(*a*,*b*,*c*,*d*,*h*) ==
formula-case (*a*, *b*,
λ*u v*. *c*(*u*, *v*, *h* ‘ *succ*(*depth*(*u*)) ‘ *u*,
h ‘ *succ*(*depth*(*v*)) ‘ *v*),
λ*u*. *d*(*u*, *h* ‘ *succ*(*depth*(*u*)) ‘ *u*))

Unfold *formula-rec* to *formula-rec-case*. Express *formula-rec* without using *rank* or *Vset*, neither of which is absolute.

lemma (**in** *M-trivial*) *formula-rec-eq*:
p ∈ *formula* ==>
formula-rec(*a*,*b*,*c*,*d*,*p*) =
transrec (*succ*(*depth*(*p*)),
λ*x h*. *Lambda* (*formula*, *formula-rec-case*(*a*,*b*,*c*,*d*,*h*))) ‘ *p*
apply (*simp add*: *formula-rec-case-def*)
apply (*induct-tac p*)

Base case for *Member*

apply (*subst transrec*, *simp add*: *formula.intros*)

Base case for *Equal*

apply (*subst transrec*, *simp add*: *formula.intros*)

Inductive step for *Nand*

apply (*subst transrec*)
apply (*simp add*: *succ-Un-distrib formula.intros*)

Inductive step for *Forall*

apply (*subst transrec*)
apply (*simp add*: *formula-imp-formula-N formula.intros*)
done

6.11.1 Absoluteness for the Formula Operator *depth*

definition
is-depth :: [*i*=>*o*,*i*,*i*] => *o* **where**

is-depth(*M*,*p*,*n*) ==
  ∃ *sn*[*M*]. ∃ *formula-n*[*M*]. ∃ *formula-sn*[*M*].
   is-formula-N(*M*,*n*,*formula-n*) & *p* ∉ *formula-n* &
   successor(*M*,*n*,*sn*) & *is-formula-N*(*M*,*sn*,*formula-sn*) & *p* ∈ *formula-sn*

lemma (**in** *M-datatypes*) *depth-abs* [*simp*]:
  [|*p* ∈ *formula*; *n* ∈ *nat*|] ==> *is-depth*(*M*,*p*,*n*) ⟷ *n* = *depth*(*p*)
apply (*subgoal-tac M*(*p*) & *M*(*n*))
prefer *2* **apply** (*blast dest*: *transM*)
apply (*simp add*: *is-depth-def*)
apply (*blast intro*: *formula-imp-formula-N nat-into-Ord formula-N-imp-eq-depth*
  dest: *formula-N-imp-depth-lt*)
done

Proof is trivial since *depth* returns natural numbers.

lemma (**in** *M-trivial*) *depth-closed* [*intro*,*simp*]:
  p ∈ *formula* ==> *M*(*depth*(*p*))
by (*simp add*: *nat-into-M*)

6.11.2 *is-formula-case*: relativization of *formula-case*

definition
is-formula-case ::
  [*i*=>*o*, [*i*,*i*,*i*]=>*o*, [*i*,*i*,*i*]=>*o*, [*i*,*i*,*i*]=>*o*, [*i*,*i*]=>*o*, *i*, *i*] => *o* **where**
— no constraint on non-formulas
is-formula-case(*M*, *is-a*, *is-b*, *is-c*, *is-d*, *p*, *z*) ==
  (∀ *x*[*M*]. ∀ *y*[*M*]. *finite-ordinal*(*M*,*x*) ⟶ *finite-ordinal*(*M*,*y*) ⟶
    is-Member(*M*,*x*,*y*,*p*) ⟶ *is-a*(*x*,*y*,*z*)) &
  (∀ *x*[*M*]. ∀ *y*[*M*]. *finite-ordinal*(*M*,*x*) ⟶ *finite-ordinal*(*M*,*y*) ⟶
    is-Equal(*M*,*x*,*y*,*p*) ⟶ *is-b*(*x*,*y*,*z*)) &
  (∀ *x*[*M*]. ∀ *y*[*M*]. *mem-formula*(*M*,*x*) ⟶ *mem-formula*(*M*,*y*) ⟶
    is-Nand(*M*,*x*,*y*,*p*) ⟶ *is-c*(*x*,*y*,*z*)) &
  (∀ *x*[*M*]. *mem-formula*(*M*,*x*) ⟶ *is-Forall*(*M*,*x*,*p*) ⟶ *is-d*(*x*,*z*))

lemma (**in** *M-datatypes*) *formula-case-abs* [*simp*]:
  [| *Relation2*(*M*,*nat*,*nat*,*is-a*,*a*); *Relation2*(*M*,*nat*,*nat*,*is-b*,*b*);
    Relation2(*M*,*formula*,*formula*,*is-c*,*c*); *Relation1*(*M*,*formula*,*is-d*,*d*);
    p ∈ *formula*; *M*(*z*) |]
  ==> *is-formula-case*(*M*,*is-a*,*is-b*,*is-c*,*is-d*,*p*,*z*) ⟷
    z = *formula-case*(*a*,*b*,*c*,*d*,*p*)
apply (*simp add*: *formula-into-M is-formula-case-def*)
apply (*erule formula.cases*)
  apply (*simp-all add*: *Relation1-def Relation2-def*)
done

lemma (**in** *M-datatypes*) *formula-case-closed* [*intro*,*simp*]:
 [|*p* ∈ *formula*;
  ∀ *x*[*M*]. ∀ *y*[*M*]. *x*∈*nat* ⟶ *y*∈*nat* ⟶ *M*(*a*(*x*,*y*));

∀ x[M]. ∀ y[M]. x∈nat ⟶ y∈nat ⟶ M(b(x,y));
∀ x[M]. ∀ y[M]. x∈formula ⟶ y∈formula ⟶ M(c(x,y));
∀ x[M]. x∈formula ⟶ M(d(x))|] ==> M(formula-case(a,b,c,d,p))
by (*erule formula.cases*, *simp-all*)

6.11.3 Absoluteness for *formula-rec*: Final Results

definition
is-formula-rec :: [i=>o, [i,i,i]=>o, i, i] => o **where**
— predicate to relativize the functional *formula-rec*
is-formula-rec(M,MH,p,z) ==
∃ dp[M]. ∃ i[M]. ∃ f[M]. *finite-ordinal*(M,dp) & *is-depth*(M,p,dp) &
successor(M,dp,i) & *fun-apply*(M,f,p,z) & *is-transrec*(M,MH,i,f)

Sufficient conditions to relativize the instance of *formula-case* in *formula-rec*

lemma (**in** *M-datatypes*) *Relation1-formula-rec-case*:
[|*Relation2*(M, nat, nat, is-a, a);
Relation2(M, nat, nat, is-b, b);
Relation2 (M, formula, formula,
is-c, λu v. c(u, v, h‘succ(depth(u))‘u, h‘succ(depth(v))‘v));
Relation1(M, formula,
is-d, λu. d(u, h ‘ succ(depth(u)) ‘ u));
M(h) |]
==> *Relation1*(M, formula,
is-formula-case (M, is-a, is-b, is-c, is-d),
formula-rec-case(a, b, c, d, h))
apply (*simp* (*no-asm*) *add*: *formula-rec-case-def Relation1-def*)
apply (*simp*)
done

This locale packages the premises of the following theorems, which is the normal purpose of locales. It doesn't accumulate constraints on the class *M*, as in most of this deveopment.

locale *Formula-Rec* = *M-eclose* +
fixes *a* **and** *is-a* **and** *b* **and** *is-b* **and** *c* **and** *is-c* **and** *d* **and** *is-d* **and** *MH*
defines
MH(u::i,f,z) ==
∀ fml[M]. is-formula(M,fml) ⟶
is-lambda
(M, fml, is-formula-case (M, is-a, is-b, is-c(f), is-d(f)), z)

assumes *a-closed*: [|x∈nat; y∈nat|] ==> M(a(x,y))
and *a-rel*: *Relation2*(M, nat, nat, is-a, a)
and *b-closed*: [|x∈nat; y∈nat|] ==> M(b(x,y))
and *b-rel*: *Relation2*(M, nat, nat, is-b, b)
and *c-closed*: [|x ∈ formula; y ∈ formula; M(gx); M(gy)|]
==> M(c(x, y, gx, gy))
and *c-rel*:
M(f) ==>

Relation2 $(M, formula, formula, is\text{-}c(f),$
$\lambda u\ v.\ c(u, v, f\ `\ succ(depth(u))\ `\ u, f\ `\ succ(depth(v))\ `\ v))$
and *d-closed*: $[|x \in formula; M(gx)|] ==> M(d(x, gx))$
and *d-rel*:
$M(f) ==>$
Relation1 $(M, formula, is\text{-}d(f), \lambda u.\ d(u, f\ `\ succ(depth(u))\ `\ u))$
and *fr-replace*: $n \in nat ==> transrec\text{-}replacement(M,MH,n)$
and *fr-lam-replace*:
$M(g) ==>$
strong-replacement
$(M, \lambda x\ y.\ x \in formula\ \&$
$y = \langle x, formula\text{-}rec\text{-}case(a,b,c,d,g,x)\rangle)$

lemma (**in** *Formula-Rec*) *formula-rec-case-closed*:
$[|M(g); p \in formula|] ==> M(formula\text{-}rec\text{-}case(a, b, c, d, g, p))$
by (*simp add*: *formula-rec-case-def a-closed b-closed c-closed d-closed*)

lemma (**in** *Formula-Rec*) *formula-rec-lam-closed*:
$M(g) ==> M(Lambda\ (formula, formula\text{-}rec\text{-}case(a,b,c,d,g)))$
by (*simp add*: *lam-closed2 fr-lam-replace formula-rec-case-closed*)

lemma (**in** *Formula-Rec*) *MH-rel2*:
relation2 $(M, MH,$
$\lambda x\ h.\ Lambda\ (formula, formula\text{-}rec\text{-}case(a,b,c,d,h)))$
apply (*simp add*: *relation2-def MH-def*, *clarify*)
apply (*rule lambda-abs2*)
apply (*rule Relation1-formula-rec-case*)
apply (*simp-all add*: *a-rel b-rel c-rel d-rel formula-rec-case-closed*)
done

lemma (**in** *Formula-Rec*) *fr-transrec-closed*:
$n \in nat$
$==> M(transrec$
$(n, \lambda x\ h.\ Lambda(formula, formula\text{-}rec\text{-}case(a, b, c, d, h))))$
by (*simp add*: *transrec-closed* [*OF fr-replace MH-rel2*]
nat-into-M formula-rec-lam-closed)

The main two results: *formula-rec* is absolute for *M*.

theorem (**in** *Formula-Rec*) *formula-rec-closed*:
$p \in formula ==> M(formula\text{-}rec(a,b,c,d,p))$
by (*simp add*: *formula-rec-eq fr-transrec-closed*
transM [*OF - formula-closed*])

theorem (**in** *Formula-Rec*) *formula-rec-abs*:
$[|\ p \in formula; M(z)|]$
$==> is\text{-}formula\text{-}rec(M,MH,p,z) \longleftrightarrow z = formula\text{-}rec(a,b,c,d,p)$
by (*simp add*: *is-formula-rec-def formula-rec-eq transM* [*OF - formula-closed*]
transrec-abs [*OF fr-replace MH-rel2*] *depth-type*
fr-transrec-closed formula-rec-lam-closed eq-commute)

end

theory *Internalizations*
  imports
    ~~/src/ZF/Constructible/Formula
    Relative Datatype-absolute
begin

6.12 Internalized Formulas for some Set-Theoretic Concepts

6.12.1 Some numbers to help write de Bruijn indices

abbreviation
  digit3 :: *i* (*3*) **where** *3* == *succ*(*2*)

abbreviation
  digit4 :: *i* (*4*) **where** *4* == *succ*(*3*)

abbreviation
  digit5 :: *i* (*5*) **where** *5* == *succ*(*4*)

abbreviation
  digit6 :: *i* (*6*) **where** *6* == *succ*(*5*)

abbreviation
  digit7 :: *i* (*7*) **where** *7* == *succ*(*6*)

abbreviation
  digit8 :: *i* (*8*) **where** *8* == *succ*(*7*)

abbreviation
  digit9 :: *i* (*9*) **where** *9* == *succ*(*8*)

6.12.2 The Empty Set, Internalized

definition
  empty-fm :: *i*=>*i* **where**
    empty-fm(*x*) == *Forall*(*Neg*(*Member*(*0*,*succ*(*x*))))

lemma *empty-type* [*TC*]:
     x ∈ *nat* ==> *empty-fm*(*x*) ∈ *formula*
by (*simp add*: *empty-fm-def*)

lemma *sats-empty-fm* [*simp*]:
   [| *x* ∈ *nat*; *env* ∈ *list*(*A*)|]
    ==> *sats*(*A*, *empty-fm*(*x*), *env*) ⟷ *empty*(##*A*, *nth*(*x*,*env*))
by (*simp add*: *empty-fm-def empty-def*)

lemma *empty-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *empty*(##*A*, *x*) ⟷ *sats*(*A*, *empty-fm*(*i*), *env*)
by *simp*

Not used. But maybe useful?

lemma *Transset-sats-empty-fm-eq-0*:
[| *n* ∈ *nat*; *env* ∈ *list*(*A*); *Transset*(*A*)|]
==> *sats*(*A*, *empty-fm*(*n*), *env*) ⟷ *nth*(*n*,*env*) = *0*
apply (*simp add*: *empty-fm-def empty-def Transset-def*, *auto*)
apply (*case-tac n* < *length*(*env*))
apply (*frule nth-type*, *assumption*+, *blast*)
apply (*simp-all add*: *not-lt-iff-le nth-eq-0*)
done

6.12.3 Unordered Pairs, Internalized

definition
upair-fm :: [*i*,*i*,*i*]=>*i* **where**
upair-fm(*x*,*y*,*z*) ==
And(*Member*(*x*,*z*),
And(*Member*(*y*,*z*),
Forall(*Implies*(*Member*(*0*,*succ*(*z*)),
Or(*Equal*(*0*,*succ*(*x*)), *Equal*(*0*,*succ*(*y*)))))))

lemma *upair-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *upair-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *upair-fm-def*)

lemma *sats-upair-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *upair-fm*(*x*,*y*,*z*), *env*) ⟷
upair(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *upair-fm-def upair-def*)

lemma *upair-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *upair*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *upair-fm*(*i*,*j*,*k*), *env*)
by (*simp*)

Useful? At least it refers to "real" unordered pairs

lemma *sats-upair-fm2* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* < *length*(*env*); *env* ∈ *list*(*A*); *Transset*(*A*)|]
==> *sats*(*A*, *upair-fm*(*x*,*y*,*z*), *env*) ⟷
nth(*z*,*env*) = {*nth*(*x*,*env*), *nth*(*y*,*env*)}
apply (*frule lt-length-in-nat*, *assumption*)

apply (*simp add*: *upair-fm-def Transset-def*, *auto*)
apply (*blast intro*: *nth-type*)
done

6.12.4 Ordered pairs, Internalized

definition
pair-fm :: [*i*,*i*,*i*]=>*i* **where**
pair-fm(*x*,*y*,*z*) ==
Exists(*And*(*upair-fm*(*succ*(*x*),*succ*(*x*),*0*),
Exists(*And*(*upair-fm*(*succ*(*succ*(*x*)),*succ*(*succ*(*y*)),*0*),
upair-fm(*1*,*0*,*succ*(*succ*(*z*)))))))

lemma *pair-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *pair-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *pair-fm-def*)

lemma *sats-pair-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *pair-fm*(*x*,*y*,*z*), *env*) ⟷
pair(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *pair-fm-def pair-def*)

lemma *pair-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *pair*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *pair-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.12.5 Binary Unions, Internalized

definition
union-fm :: [*i*,*i*,*i*]=>*i* **where**
union-fm(*x*,*y*,*z*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*z*)),
Or(*Member*(*0*,*succ*(*x*)),*Member*(*0*,*succ*(*y*)))))

lemma *union-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *union-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *union-fm-def*)

lemma *sats-union-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *union-fm*(*x*,*y*,*z*), *env*) ⟷
union(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *union-fm-def union-def*)

lemma *union-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]

==> *union*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *union-fm*(*i*,*j*,*k*), *env*)
by (*simp*)

6.12.6 Set "Cons," Internalized

definition
cons-fm :: [*i*,*i*,*i*]=>*i* **where**
cons-fm(*x*,*y*,*z*) ==
Exists(*And*(*upair-fm*(*succ*(*x*),*succ*(*x*),*0*),
union-fm(*0*,*succ*(*y*),*succ*(*z*))))

lemma *cons-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *cons-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *cons-fm-def*)

lemma *sats-cons-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *cons-fm*(*x*,*y*,*z*), *env*) ⟷
is-cons(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *cons-fm-def is-cons-def*)

lemma *cons-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-cons*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *cons-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.12.7 Successor Function, Internalized

definition
succ-fm :: [*i*,*i*]=>*i* **where**
succ-fm(*x*,*y*) == *cons-fm*(*x*,*x*,*y*)

lemma *succ-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *succ-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *succ-fm-def*)

lemma *sats-succ-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *succ-fm*(*x*,*y*), *env*) ⟷
successor(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *succ-fm-def successor-def*)

lemma *successor-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *successor*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *succ-fm*(*i*,*j*), *env*)
by *simp*

6.12.8 The Number 1, Internalized

definition
  number1-fm :: *i=>i* **where**
    number1-fm(a) == Exists(And(empty-fm(0), succ-fm(0,succ(a))))

lemma *number1-type* [*TC*]:
     x ∈ nat ==> number1-fm(x) ∈ formula
by (*simp add*: *number1-fm-def*)

lemma *sats-number1-fm* [*simp*]:
   [| *x ∈ nat*; *env ∈ list(A)*|]
    ==> *sats(A, number1-fm(x), env)* ⟷ *number1(##A, nth(x,env))*
by (*simp add*: *number1-fm-def number1-def*)

lemma *number1-iff-sats*:
      [| *nth(i,env) = x*; *nth(j,env) = y*;
          i ∈ nat; *env ∈ list(A)*|]
       ==> *number1(##A, x)* ⟷ *sats(A, number1-fm(i), env)*
by *simp*

6.12.9 Big Union, Internalized

definition
  big-union-fm :: [*i,i*]=>*i* **where**
    big-union-fm(A,z) ==
       Forall(Iff(Member(0,succ(z)),
                 Exists(And(Member(0,succ(succ(A))), Member(1,0)))))

lemma *big-union-type* [*TC*]:
     [| *x ∈ nat*; *y ∈ nat* |] ==> *big-union-fm(x,y) ∈ formula*
by (*simp add*: *big-union-fm-def*)

lemma *sats-big-union-fm* [*simp*]:
   [| *x ∈ nat*; *y ∈ nat*; *env ∈ list(A)*|]
    ==> *sats(A, big-union-fm(x,y), env)* ⟷
        big-union(##A, nth(x,env), nth(y,env))
by (*simp add*: *big-union-fm-def big-union-def*)

lemma *big-union-iff-sats*:
      [| *nth(i,env) = x*; *nth(j,env) = y*;
          i ∈ nat; *j ∈ nat*; *env ∈ list(A)*|]
       ==> *big-union(##A, x, y)* ⟷ *sats(A, big-union-fm(i,j), env)*
by *simp*

6.12.10 Variants of Satisfaction Definitions for Ordinals, etc.

The *sats* theorems below are standard versions of the ones proved in theory *Formula*. They relate elements of type *formula* to relativized concepts such as *subset* or *ordinal* rather than to real concepts such as *Ord*. Now that we have instantiated the locale *M-trivial*, we no longer require the earlier versions.

lemma *sats-subset-fm′*:
  [|*x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
   ==> *sats*(*A*, *subset-fm*(*x*,*y*), *env*) ⟷ *subset*(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *subset-fm-def Relative.subset-def*)

lemma *sats-transset-fm′*:
  [|*x* ∈ *nat*; *env* ∈ *list*(*A*)|]
   ==> *sats*(*A*, *transset-fm*(*x*), *env*) ⟷ *transitive-set*(##*A*, *nth*(*x*,*env*))
by (*simp add*: *sats-subset-fm′ transset-fm-def transitive-set-def*)

lemma *sats-ordinal-fm′*:
  [|*x* ∈ *nat*; *env* ∈ *list*(*A*)|]
   ==> *sats*(*A*, *ordinal-fm*(*x*), *env*) ⟷ *ordinal*(##*A*,*nth*(*x*,*env*))
by (*simp add*: *sats-transset-fm′ ordinal-fm-def ordinal-def*)

lemma *ordinal-iff-sats*:
    [| *nth*(*i*,*env*) = *x*; *i* ∈ *nat*; *env* ∈ *list*(*A*)|]
     ==> *ordinal*(##*A*, *x*) ⟷ *sats*(*A*, *ordinal-fm*(*i*), *env*)
by (*simp add*: *sats-ordinal-fm′*)

6.12.11 Membership Relation, Internalized

definition
 Memrel-fm :: [*i*,*i*]=>*i* **where**
  Memrel-fm(*A*,*r*) ==
    Forall(*Iff*(*Member*(*0*,*succ*(*r*)),
         Exists(*And*(*Member*(*0*,*succ*(*succ*(*A*))),
           Exists(*And*(*Member*(*0*,*succ*(*succ*(*succ*(*A*)))),
             And(*Member*(*1*,*0*),
               pair-fm(*1*,*0*,*2*))))))))

lemma *Memrel-type* [*TC*]:
   [| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *Memrel-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *Memrel-fm-def*)

lemma *sats-Memrel-fm* [*simp*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
   ==> *sats*(*A*, *Memrel-fm*(*x*,*y*), *env*) ⟷
     membership(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *Memrel-fm-def membership-def*)

lemma *Memrel-iff-sats*:
    [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
      i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
     ==> *membership*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *Memrel-fm*(*i*,*j*), *env*)
by *simp*

6.12.12 Predecessor Set, Internalized

definition
pred-set-fm :: [*i*,*i*,*i*,*i*]=>*i* **where**
pred-set-fm(*A*,*x*,*r*,*B*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*B*)),
Exists(*And*(*Member*(*0*,*succ*(*succ*(*r*))),
And(*Member*(*1*,*succ*(*succ*(*A*))),
pair-fm(*1*,*succ*(*succ*(*x*)),*0*))))))

lemma *pred-set-type* [*TC*]:
[| *A* ∈ *nat*; *x* ∈ *nat*; *r* ∈ *nat*; *B* ∈ *nat* |]
==> *pred-set-fm*(*A*,*x*,*r*,*B*) ∈ *formula*
by (*simp add*: *pred-set-fm-def*)

lemma *sats-pred-set-fm* [*simp*]:
[| *U* ∈ *nat*; *x* ∈ *nat*; *r* ∈ *nat*; *B* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *pred-set-fm*(*U*,*x*,*r*,*B*), *env*) ⟷
pred-set(##*A*, *nth*(*U*,*env*), *nth*(*x*,*env*), *nth*(*r*,*env*), *nth*(*B*,*env*))
by (*simp add*: *pred-set-fm-def pred-set-def*)

lemma *pred-set-iff-sats*:
[| *nth*(*i*,*env*) = *U*; *nth*(*j*,*env*) = *x*; *nth*(*k*,*env*) = *r*; *nth*(*l*,*env*) = *B*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *l* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *pred-set*(##*A*,*U*,*x*,*r*,*B*) ⟷ *sats*(*A*, *pred-set-fm*(*i*,*j*,*k*,*l*), *env*)
by (*simp*)

6.12.13 Domain of a Relation, Internalized

definition
domain-fm :: [*i*,*i*]=>*i* **where**
domain-fm(*r*,*z*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*z*)),
Exists(*And*(*Member*(*0*,*succ*(*succ*(*r*))),
Exists(*pair-fm*(*2*,*0*,*1*))))))

lemma *domain-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *domain-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *domain-fm-def*)

lemma *sats-domain-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *domain-fm*(*x*,*y*), *env*) ⟷
is-domain(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *domain-fm-def is-domain-def*)

lemma *domain-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]

==> *is-domain*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *domain-fm*(*i*,*j*), *env*)
by *simp*

6.12.14 Range of a Relation, Internalized

definition
range-fm :: [*i*,*i*]=>*i* **where**
range-fm(*r*,*z*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*z*)),
Exists(*And*(*Member*(*0*,*succ*(*succ*(*r*))),
Exists(*pair-fm*(*0*,*2*,*1*))))))

lemma *range-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *range-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *range-fm-def*)

lemma *sats-range-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *range-fm*(*x*,*y*), *env*) ⟷
is-range(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *range-fm-def is-range-def*)

lemma *range-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-range*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *range-fm*(*i*,*j*), *env*)
by *simp*

6.12.15 Field of a Relation, Internalized

definition
field-fm :: [*i*,*i*]=>*i* **where**
field-fm(*r*,*z*) ==
Exists(*And*(*domain-fm*(*succ*(*r*),*0*),
Exists(*And*(*range-fm*(*succ*(*succ*(*r*)),*0*),
union-fm(*1*,*0*,*succ*(*succ*(*z*)))))))

lemma *field-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *field-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *field-fm-def*)

lemma *sats-field-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *field-fm*(*x*,*y*), *env*) ⟷
is-field(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *field-fm-def is-field-def*)

lemma *field-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]

==> *is-field*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *field-fm*(*i*,*j*), *env*)
by *simp*

6.12.16 Image under a Relation, Internalized

definition
image-fm :: [*i*,*i*,*i*]=>*i* **where**
image-fm(*r*,*A*,*z*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*z*)),
Exists(*And*(*Member*(*0*,*succ*(*succ*(*r*))),
Exists(*And*(*Member*(*0*,*succ*(*succ*(*succ*(*A*)))),
pair-fm(*0*,*2*,*1*)))))))

lemma *image-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *image-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *image-fm-def*)

lemma *sats-image-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *image-fm*(*x*,*y*,*z*), *env*) ⟷
image(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *image-fm-def Relative.image-def*)

lemma *image-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *image*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *image-fm*(*i*,*j*,*k*), *env*)
by (*simp*)

6.12.17 Pre-Image under a Relation, Internalized

definition
pre-image-fm :: [*i*,*i*,*i*]=>*i* **where**
pre-image-fm(*r*,*A*,*z*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*z*)),
Exists(*And*(*Member*(*0*,*succ*(*succ*(*r*))),
Exists(*And*(*Member*(*0*,*succ*(*succ*(*succ*(*A*)))),
pair-fm(*2*,*0*,*1*)))))))

lemma *pre-image-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *pre-image-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *pre-image-fm-def*)

lemma *sats-pre-image-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *pre-image-fm*(*x*,*y*,*z*), *env*) ⟷
pre-image(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *pre-image-fm-def Relative.pre-image-def*)

lemma *pre-image-iff-sats*:

```
      [| nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
          i ∈ nat; j ∈ nat; k ∈ nat; env ∈ list(A)|]
       ==> pre-image(##A, x, y, z) ⟷ sats(A, pre-image-fm(i,j,k), env)
by (simp)
```

6.12.18 Function Application, Internalized

```
definition
  fun-apply-fm :: [i,i,i]=>i where
    fun-apply-fm(f,x,y) ==
        Exists(Exists(And(upair-fm(succ(succ(x)), succ(succ(x)), 1),
                         And(image-fm(succ(succ(f)), 1, 0),
                             big-union-fm(0,succ(succ(y)))))))

lemma fun-apply-type [TC]:
     [| x ∈ nat; y ∈ nat; z ∈ nat |] ==> fun-apply-fm(x,y,z) ∈ formula
by (simp add: fun-apply-fm-def)

lemma sats-fun-apply-fm [simp]:
   [| x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, fun-apply-fm(x,y,z), env) ⟷
         fun-apply(##A, nth(x,env), nth(y,env), nth(z,env))
by (simp add: fun-apply-fm-def fun-apply-def)

lemma fun-apply-iff-sats:
      [| nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
          i ∈ nat; j ∈ nat; k ∈ nat; env ∈ list(A)|]
       ==> fun-apply(##A, x, y, z) ⟷ sats(A, fun-apply-fm(i,j,k), env)
by simp
```

6.12.19 The Concept of Relation, Internalized

```
definition
  relation-fm :: i=>i where
    relation-fm(r) ==
       Forall(Implies(Member(0,succ(r)), Exists(Exists(pair-fm(1,0,2)))))

lemma relation-type [TC]:
     [| x ∈ nat |] ==> relation-fm(x) ∈ formula
by (simp add: relation-fm-def)

lemma sats-relation-fm [simp]:
   [| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, relation-fm(x), env) ⟷ is-relation(##A, nth(x,env))
by (simp add: relation-fm-def is-relation-def)

lemma relation-iff-sats:
      [| nth(i,env) = x; nth(j,env) = y;
          i ∈ nat; env ∈ list(A)|]
       ==> is-relation(##A, x) ⟷ sats(A, relation-fm(i), env)
```

by *simp*

6.12.20 The Concept of Function, Internalized

definition
function-fm :: *i=>i* **where**
function-fm(r) ==
Forall(Forall(Forall(Forall(Forall(
Implies(pair-fm(4,3,1),
Implies(pair-fm(4,2,0),
Implies(Member(1,r#+5),
Implies(Member(0,r#+5), Equal(3,2))))))))))

lemma *function-type* [*TC*]:
[| *x* ∈ *nat* |] ==> *function-fm(x)* ∈ *formula*
by (*simp add*: *function-fm-def*)

lemma *sats-function-fm* [*simp*]:
[| *x* ∈ *nat*; *env* ∈ *list(A)*|]
==> *sats(A, function-fm(x), env)* ⟷ *is-function(##A, nth(x,env))*
by (*simp add*: *function-fm-def is-function-def*)

lemma *is-function-iff-sats*:
[| *nth(i,env)* = *x*; *nth(j,env)* = *y*;
i ∈ *nat*; *env* ∈ *list(A)*|]
==> *is-function(##A, x)* ⟷ *sats(A, function-fm(i), env)*
by *simp*

6.12.21 Typed Functions, Internalized

definition
typed-function-fm :: [*i,i,i*]=>*i* **where**
typed-function-fm(A,B,r) ==
And(function-fm(r),
And(relation-fm(r),
And(domain-fm(r,A),
Forall(Implies(Member(0,succ(r)),
Forall(Forall(Implies(pair-fm(1,0,2),Member(0,B#+3)))))))))

lemma *typed-function-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *typed-function-fm(x,y,z)* ∈ *formula*
by (*simp add*: *typed-function-fm-def*)

lemma *sats-typed-function-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list(A)*|]
==> *sats(A, typed-function-fm(x,y,z), env)* ⟷
typed-function(##A, nth(x,env), nth(y,env), nth(z,env))
by (*simp add*: *typed-function-fm-def typed-function-def*)

lemma *typed-function-iff-sats*:

[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
    i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *typed-function*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *typed-function-fm*(*i*,*j*,*k*), *env*)
by *simp*

lemmas *function-iff-sats* =
  empty-iff-sats number1-iff-sats
  upair-iff-sats pair-iff-sats union-iff-sats
  big-union-iff-sats cons-iff-sats successor-iff-sats
  fun-apply-iff-sats Memrel-iff-sats
  pred-set-iff-sats domain-iff-sats range-iff-sats field-iff-sats
  image-iff-sats pre-image-iff-sats
  relation-iff-sats is-function-iff-sats

6.12.22 Composition of Relations, Internalized

definition
  composition-fm :: [*i*,*i*,*i*]=>*i* **where**
  composition-fm(*r*,*s*,*t*) ==
    Forall(*Iff*(*Member*(*0*,*succ*(*t*)),
      Exists(*Exists*(*Exists*(*Exists*(*Exists*(
        And(*pair-fm*(*4*,*2*,*5*),
         And(*pair-fm*(*4*,*3*,*1*),
          And(*pair-fm*(*3*,*2*,*0*),
           And(*Member*(*1*,*s*#+*6*), *Member*(*0*,*r*#+*6*))))))))))))

lemma *composition-type* [*TC*]:
    [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *composition-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *composition-fm-def*)

lemma *sats-composition-fm* [*simp*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
   ==> *sats*(*A*, *composition-fm*(*x*,*y*,*z*), *env*) ⟷
     composition(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *composition-fm-def composition-def*)

lemma *composition-iff-sats*:
    [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
      i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
    ==> *composition*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *composition-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.12.23 Injections, Internalized

definition
  injection-fm :: [*i*,*i*,*i*]=>*i* **where**
  injection-fm(*A*,*B*,*f*) ==
   And(*typed-function-fm*(*A*,*B*,*f*),
     Forall(*Forall*(*Forall*(*Forall*(*Forall*(
       Implies(*pair-fm*(*4*,*2*,*1*),

Implies(*pair-fm*(*3*,*2*,*0*),
  Implies(*Member*(*1*,*f*#+*5*),
    Implies(*Member*(*0*,*f*#+*5*), *Equal*(*4*,*3*)))))))))))

lemma *injection-type* [*TC*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *injection-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *injection-fm-def*)

lemma *sats-injection-fm* [*simp*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *injection-fm*(*x*,*y*,*z*), *env*) ⟷
    injection(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *injection-fm-def injection-def*)

lemma *injection-iff-sats*:
  [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
    i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *injection*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *injection-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.12.24 Surjections, Internalized

definition
  surjection-fm :: [*i*,*i*,*i*]=>*i* **where**
  surjection-fm(*A*,*B*,*f*) ==
   And(*typed-function-fm*(*A*,*B*,*f*),
    Forall(*Implies*(*Member*(*0*,*succ*(*B*)),
      Exists(*And*(*Member*(*0*,*succ*(*succ*(*A*))),
        fun-apply-fm(*succ*(*succ*(*f*)),*0*,*1*))))))

lemma *surjection-type* [*TC*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *surjection-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *surjection-fm-def*)

lemma *sats-surjection-fm* [*simp*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *surjection-fm*(*x*,*y*,*z*), *env*) ⟷
    surjection(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *surjection-fm-def surjection-def*)

lemma *surjection-iff-sats*:
  [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
    i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *surjection*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *surjection-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.12.25 Bijections, Internalized

definition

bijection-fm :: [*i*,*i*,*i*]=>*i* **where**
bijection-fm(*A*,*B*,*f*) == *And*(*injection-fm*(*A*,*B*,*f*), *surjection-fm*(*A*,*B*,*f*))

lemma *bijection-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *bijection-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *bijection-fm-def*)

lemma *sats-bijection-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *bijection-fm*(*x*,*y*,*z*), *env*) ⟷
bijection(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *bijection-fm-def bijection-def*)

lemma *bijection-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *bijection*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *bijection-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.12.26 Restriction of a Relation, Internalized

definition
restriction-fm :: [*i*,*i*,*i*]=>*i* **where**
restriction-fm(*r*,*A*,*z*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*z*)),
And(*Member*(*0*,*succ*(*r*)),
Exists(*And*(*Member*(*0*,*succ*(*succ*(*A*))),
Exists(*pair-fm*(*1*,*0*,*2*)))))))

lemma *restriction-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *restriction-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *restriction-fm-def*)

lemma *sats-restriction-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *restriction-fm*(*x*,*y*,*z*), *env*) ⟷
restriction(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *restriction-fm-def restriction-def*)

lemma *restriction-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *restriction*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *restriction-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.12.27 Order-Isomorphisms, Internalized

definition
order-isomorphism-fm :: [*i*,*i*,*i*,*i*,*i*]=>*i* **where**
order-isomorphism-fm(*A*,*r*,*B*,*s*,*f*) ==

And(*bijection-fm*(*A*,*B*,*f*),
  Forall(*Implies*(*Member*(*0*,*succ*(*A*)),
    Forall(*Implies*(*Member*(*0*,*succ*(*succ*(*A*))),
      Forall(*Forall*(*Forall*(*Forall*(
        Implies(*pair-fm*(*5*,*4*,*3*),
          Implies(*fun-apply-fm*(*f* #+ *6*,*5*,*2*),
            Implies(*fun-apply-fm*(*f* #+ *6*,*4*,*1*),
              Implies(*pair-fm*(*2*,*1*,*0*),
                Iff(*Member*(*3*,*r* #+ *6*), *Member*(*0*,*s* #+ *6*)))))))))))))))

lemma *order-isomorphism-type* [*TC*]:
  [| *A* ∈ *nat*; *r* ∈ *nat*; *B* ∈ *nat*; *s* ∈ *nat*; *f* ∈ *nat* |]
  ==> *order-isomorphism-fm*(*A*,*r*,*B*,*s*,*f*) ∈ *formula*
by (*simp add*: *order-isomorphism-fm-def*)

lemma *sats-order-isomorphism-fm* [*simp*]:
  [| *U* ∈ *nat*; *r* ∈ *nat*; *B* ∈ *nat*; *s* ∈ *nat*; *f* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *order-isomorphism-fm*(*U*,*r*,*B*,*s*,*f*), *env*) ⟷
    order-isomorphism(##*A*, *nth*(*U*,*env*), *nth*(*r*,*env*), *nth*(*B*,*env*),
      nth(*s*,*env*), *nth*(*f*,*env*))
by (*simp add*: *order-isomorphism-fm-def order-isomorphism-def*)

lemma *order-isomorphism-iff-sats*:
  [| *nth*(*i*,*env*) = *U*; *nth*(*j*,*env*) = *r*; *nth*(*k*,*env*) = *B*; *nth*(*j'*,*env*) = *s*;
    nth(*k'*,*env*) = *f*;
    i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *j'* ∈ *nat*; *k'* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *order-isomorphism*(##*A*,*U*,*r*,*B*,*s*,*f*) ⟷
    sats(*A*, *order-isomorphism-fm*(*i*,*j*,*k*,*j'*,*k'*), *env*)
by *simp*

6.12.28 Limit Ordinals, Internalized

A limit ordinal is a non-empty, successor-closed ordinal

definition
  limit-ordinal-fm :: *i*=>*i* **where**
  limit-ordinal-fm(*x*) ==
    And(*ordinal-fm*(*x*),
      And(*Neg*(*empty-fm*(*x*)),
        Forall(*Implies*(*Member*(*0*,*succ*(*x*)),
          Exists(*And*(*Member*(*0*,*succ*(*succ*(*x*))),
            succ-fm(*1*,*0*)))))))

lemma *limit-ordinal-type* [*TC*]:
  x ∈ *nat* ==> *limit-ordinal-fm*(*x*) ∈ *formula*
by (*simp add*: *limit-ordinal-fm-def*)

lemma *sats-limit-ordinal-fm* [*simp*]:
  [| *x* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *limit-ordinal-fm*(*x*), *env*) ⟷ *limit-ordinal*(##*A*, *nth*(*x*,*env*))

by (*simp add*: *limit-ordinal-fm-def limit-ordinal-def sats-ordinal-fm′*)

lemma *limit-ordinal-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *limit-ordinal*(##*A*, *x*) ⟷ *sats*(*A*, *limit-ordinal-fm*(*i*), *env*)
by *simp*

6.12.29 Finite Ordinals: The Predicate "Is A Natural Number"

definition
finite-ordinal-fm :: *i*=>*i* **where**
finite-ordinal-fm(*x*) ==
And(*ordinal-fm*(*x*),
And(*Neg*(*limit-ordinal-fm*(*x*)),
Forall(*Implies*(*Member*(*0*,*succ*(*x*)),
Neg(*limit-ordinal-fm*(*0*))))))

lemma *finite-ordinal-type* [*TC*]:
x ∈ *nat* ==> *finite-ordinal-fm*(*x*) ∈ *formula*
by (*simp add*: *finite-ordinal-fm-def*)

lemma *sats-finite-ordinal-fm* [*simp*]:
[| *x* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *finite-ordinal-fm*(*x*), *env*) ⟷ *finite-ordinal*(##*A*, *nth*(*x*,*env*))
by (*simp add*: *finite-ordinal-fm-def sats-ordinal-fm′ finite-ordinal-def*)

lemma *finite-ordinal-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *finite-ordinal*(##*A*, *x*) ⟷ *sats*(*A*, *finite-ordinal-fm*(*i*), *env*)
by *simp*

6.12.30 Omega: The Set of Natural Numbers

definition
omega-fm :: *i*=>*i* **where**
omega-fm(*x*) ==
And(*limit-ordinal-fm*(*x*),
Forall(*Implies*(*Member*(*0*,*succ*(*x*)),
Neg(*limit-ordinal-fm*(*0*)))))

lemma *omega-type* [*TC*]:
x ∈ *nat* ==> *omega-fm*(*x*) ∈ *formula*
by (*simp add*: *omega-fm-def*)

lemma *sats-omega-fm* [*simp*]:
[| *x* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *omega-fm*(*x*), *env*) ⟷ *omega*(##*A*, *nth*(*x*,*env*))
by (*simp add*: *omega-fm-def omega-def*)

lemma *omega-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *omega*(##*A*, *x*) ⟷ *sats*(*A*, *omega-fm*(*i*), *env*)
by *simp*

lemmas *fun-plus-iff-sats* =
typed-function-iff-sats composition-iff-sats
injection-iff-sats surjection-iff-sats
bijection-iff-sats restriction-iff-sats
order-isomorphism-iff-sats finite-ordinal-iff-sats
ordinal-iff-sats limit-ordinal-iff-sats omega-iff-sats

6.13 Internalized Forms of Data Structuring Operators

6.13.1 The Formula *is-Inl*, Internalized

definition
Inl-fm :: [*i*,*i*]=>*i* **where**
Inl-fm(*a*,*z*) == *Exists*(*And*(*empty-fm*(*0*), *pair-fm*(*0*,*succ*(*a*),*succ*(*z*))))

lemma *Inl-type* [*TC*]:
[| *x* ∈ *nat*; *z* ∈ *nat* |] ==> *Inl-fm*(*x*,*z*) ∈ *formula*
by (*simp add*: *Inl-fm-def*)

lemma *sats-Inl-fm* [*simp*]:
[| *x* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *Inl-fm*(*x*,*z*), *env*) ⟷ *is-Inl*(##*A*, *nth*(*x*,*env*), *nth*(*z*,*env*))

by (*simp add*: *Inl-fm-def is-Inl-def*)

lemma *Inl-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-Inl*(##*A*, *x*, *z*) ⟷ *sats*(*A*, *Inl-fm*(*i*,*k*), *env*)
by *simp*

6.13.2 The Formula *is-Inr*, Internalized

definition
Inr-fm :: [*i*,*i*]=>*i* **where**
Inr-fm(*a*,*z*) == *Exists*(*And*(*number1-fm*(*0*), *pair-fm*(*0*,*succ*(*a*),*succ*(*z*))))

lemma *Inr-type* [*TC*]:
[| *x* ∈ *nat*; *z* ∈ *nat* |] ==> *Inr-fm*(*x*,*z*) ∈ *formula*
by (*simp add*: *Inr-fm-def*)

lemma *sats-Inr-fm* [*simp*]:
[| *x* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]

==> *sats*(*A*, *Inr-fm*(*x*,*z*), *env*) ⟷ *is-Inr*(##*A*, *nth*(*x*,*env*), *nth*(*z*,*env*))
by (*simp add*: *Inr-fm-def is-Inr-def*)

lemma *Inr-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-Inr*(##*A*, *x*, *z*) ⟷ *sats*(*A*, *Inr-fm*(*i*,*k*), *env*)
by *simp*

6.13.3 The Formula *is-Nil*, Internalized

definition
Nil-fm :: *i*=>*i* **where**
Nil-fm(*x*) == *Exists*(*And*(*empty-fm*(*0*), *Inl-fm*(*0*,*succ*(*x*))))

lemma *Nil-type* [*TC*]: *x* ∈ *nat* ==> *Nil-fm*(*x*) ∈ *formula*
by (*simp add*: *Nil-fm-def*)

lemma *sats-Nil-fm* [*simp*]:
[| *x* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *Nil-fm*(*x*), *env*) ⟷ *is-Nil*(##*A*, *nth*(*x*,*env*))
by (*simp add*: *Nil-fm-def is-Nil-def*)

lemma *Nil-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *i* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-Nil*(##*A*, *x*) ⟷ *sats*(*A*, *Nil-fm*(*i*), *env*)
by *simp*

6.13.4 The Formula *is-Cons*, Internalized

definition
Cons-fm :: [*i*,*i*,*i*]=>*i* **where**
Cons-fm(*a*,*l*,*Z*) ==
Exists(*And*(*pair-fm*(*succ*(*a*),*succ*(*l*),*0*), *Inr-fm*(*0*,*succ*(*Z*))))

lemma *Cons-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *Cons-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *Cons-fm-def*)

lemma *sats-Cons-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *Cons-fm*(*x*,*y*,*z*), *env*) ⟷
is-Cons(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *Cons-fm-def is-Cons-def*)

lemma *Cons-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==>*is-Cons*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *Cons-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.13.5 The Formula *is-quasilist*, Internalized

```
definition
  quasilist-fm :: i=>i where
    quasilist-fm(x) ==
      Or(Nil-fm(x), Exists(Exists(Cons-fm(1,0,succ(succ(x))))))

lemma quasilist-type [TC]: x ∈ nat ==> quasilist-fm(x) ∈ formula
by (simp add: quasilist-fm-def)

lemma sats-quasilist-fm [simp]:
   [| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, quasilist-fm(x), env) ⟷ is-quasilist(##A, nth(x,env))
by (simp add: quasilist-fm-def is-quasilist-def)

lemma quasilist-iff-sats:
      [| nth(i,env) = x; i ∈ nat; env ∈ list(A)|]
       ==> is-quasilist(##A, x) ⟷ sats(A, quasilist-fm(i), env)
by simp
```

6.14 Absoluteness for the Function *nth*

6.14.1 The Formula *is-hd*, Internalized

```
definition
  hd-fm :: [i,i]=>i where
    hd-fm(xs,H) ==
      And(Implies(Nil-fm(xs), empty-fm(H)),
          And(Forall(Forall(Or(Neg(Cons-fm(1,0,xs#+2)), Equal(H#+2,1)))),
              Or(quasilist-fm(xs), empty-fm(H))))

lemma hd-type [TC]:
     [| x ∈ nat; y ∈ nat |] ==> hd-fm(x,y) ∈ formula
by (simp add: hd-fm-def)

lemma sats-hd-fm [simp]:
   [| x ∈ nat; y ∈ nat; env ∈ list(A)|]
    ==> sats(A, hd-fm(x,y), env) ⟷ is-hd(##A, nth(x,env), nth(y,env))
by (simp add: hd-fm-def is-hd-def)

lemma hd-iff-sats:
      [| nth(i,env) = x; nth(j,env) = y;
          i ∈ nat; j ∈ nat; env ∈ list(A)|]
       ==> is-hd(##A, x, y) ⟷ sats(A, hd-fm(i,j), env)
by simp
```

6.14.2 The Formula *is-tl*, Internalized

```
definition
  tl-fm :: [i,i]=>i where
```

tl-fm(*xs*,*T*) ==
  And(*Implies*(*Nil-fm*(*xs*), *Equal*(*T*,*xs*)),
    And(*Forall*(*Forall*(*Or*(*Neg*(*Cons-fm*(*1*,*0*,*xs*#+*2*)), *Equal*(*T*#+*2*,*0*)))),
      Or(*quasilist-fm*(*xs*), *empty-fm*(*T*))))

lemma *tl-type* [*TC*]:
  [| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *tl-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *tl-fm-def*)

lemma *sats-tl-fm* [*simp*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *tl-fm*(*x*,*y*), *env*) ⟷ *is-tl*(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *tl-fm-def is-tl-def*)

lemma *tl-iff-sats*:
  [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
    i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *is-tl*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *tl-fm*(*i*,*j*), *env*)
by *simp*

6.14.3 The Operator *is-bool-of-o*

The formula p has no free variables.

definition
  bool-of-o-fm :: [*i*, *i*]=>*i* **where**
  bool-of-o-fm(*p*,*z*) ==
    Or(*And*(*p*,*number1-fm*(*z*)),
      And(*Neg*(*p*),*empty-fm*(*z*)))

lemma *is-bool-of-o-type* [*TC*]:
  [| *p* ∈ *formula*; *z* ∈ *nat* |] ==> *bool-of-o-fm*(*p*,*z*) ∈ *formula*
by (*simp add*: *bool-of-o-fm-def*)

lemma *sats-bool-of-o-fm*:
  assumes *p-iff-sats*: *P* ⟷ *sats*(*A*, *p*, *env*)
  shows
    [|*z* ∈ *nat*; *env* ∈ *list*(*A*)|]
    ==> *sats*(*A*, *bool-of-o-fm*(*p*,*z*), *env*) ⟷
      is-bool-of-o(##*A*, *P*, *nth*(*z*,*env*))
by (*simp add*: *bool-of-o-fm-def is-bool-of-o-def p-iff-sats* [*THEN iff-sym*])

lemma *is-bool-of-o-iff-sats*:
  [| *P* ⟷ *sats*(*A*, *p*, *env*); *nth*(*k*,*env*) = *z*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *is-bool-of-o*(##*A*, *P*, *z*) ⟷ *sats*(*A*, *bool-of-o-fm*(*p*,*k*), *env*)
by (*simp add*: *sats-bool-of-o-fm*)

6.15 More Internalizations

6.15.1 The Operator *is-lambda*

The two arguments of *p* are always 1, 0. Remember that *p* will be enclosed by three quantifiers.

```
definition
  lambda-fm :: [i, i, i]=>i where
  lambda-fm(p,A,z) ==
    Forall(Iff(Member(0,succ(z)),
           Exists(Exists(And(Member(1,A#+3),
                         And(pair-fm(1,0,2), p))))))
```

We call *p* with arguments x, y by equating them with the corresponding quantified variables with de Bruijn indices 1, 0.

```
lemma is-lambda-type [TC]:
     [| p ∈ formula; x ∈ nat; y ∈ nat |]
      ==> lambda-fm(p,x,y) ∈ formula
by (simp add: lambda-fm-def)

lemma sats-lambda-fm:
  assumes is-b-iff-sats:
     !!a0 a1 a2.
        [|a0∈A; a1∈A; a2∈A|]
        ==> is-b(a1, a0) ⟷ sats(A, p, Cons(a0,Cons(a1,Cons(a2,env))))
  shows
     [|x ∈ nat; y ∈ nat; env ∈ list(A)|]
      ==> sats(A, lambda-fm(p,x,y), env) ⟷
          is-lambda(##A, nth(x,env), is-b, nth(y,env))
by (simp add: lambda-fm-def is-lambda-def is-b-iff-sats [THEN iff-sym])
```

6.15.2 The Operator *is-Member*, Internalized

```
definition
  Member-fm :: [i,i,i]=>i where
    Member-fm(x,y,Z) ==
       Exists(Exists(And(pair-fm(x#+2,y#+2,1),
                    And(Inl-fm(1,0), Inl-fm(0,Z#+2)))))

lemma is-Member-type [TC]:
     [| x ∈ nat; y ∈ nat; z ∈ nat |] ==> Member-fm(x,y,z) ∈ formula
by (simp add: Member-fm-def)

lemma sats-Member-fm [simp]:
   [| x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, Member-fm(x,y,z), env) ⟷
        is-Member(##A, nth(x,env), nth(y,env), nth(z,env))
by (simp add: Member-fm-def is-Member-def)
```

lemma *Member-iff-sats*:
  [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
   i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *is-Member*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *Member-fm*(*i*,*j*,*k*), *env*)
by (*simp add*: *sats-Member-fm*)

6.15.3 The Operator *is-Equal*, Internalized

definition
 Equal-fm :: [*i*,*i*,*i*]=>*i* **where**
  Equal-fm(*x*,*y*,*Z*) ==
   Exists(*Exists*(*And*(*pair-fm*(*x*#+*2*,*y*#+*2*,*1*),
       And(*Inr-fm*(*1*,*0*), *Inl-fm*(*0*,*Z*#+*2*)))))

lemma *is-Equal-type* [*TC*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *Equal-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *Equal-fm-def*)

lemma *sats-Equal-fm* [*simp*]:
 [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *Equal-fm*(*x*,*y*,*z*), *env*) ⟷
   is-Equal(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *Equal-fm-def is-Equal-def*)

lemma *Equal-iff-sats*:
  [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
   i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *is-Equal*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *Equal-fm*(*i*,*j*,*k*), *env*)
by (*simp add*: *sats-Equal-fm*)

6.15.4 The Operator *is-Nand*, Internalized

definition
 Nand-fm :: [*i*,*i*,*i*]=>*i* **where**
  Nand-fm(*x*,*y*,*Z*) ==
   Exists(*Exists*(*And*(*pair-fm*(*x*#+*2*,*y*#+*2*,*1*),
       And(*Inl-fm*(*1*,*0*), *Inr-fm*(*0*,*Z*#+*2*)))))

lemma *is-Nand-type* [*TC*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *Nand-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *Nand-fm-def*)

lemma *sats-Nand-fm* [*simp*]:
 [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *Nand-fm*(*x*,*y*,*z*), *env*) ⟷
   is-Nand(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *Nand-fm-def is-Nand-def*)

lemma *Nand-iff-sats*:
  [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;

i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-Nand*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *Nand-fm*(*i*,*j*,*k*), *env*)
by (*simp add*: *sats-Nand-fm*)

6.15.5 The Operator *is-Forall*, Internalized

definition
Forall-fm :: [*i*,*i*]=>*i* **where**
Forall-fm(*x*,*Z*) ==
Exists(*And*(*Inr-fm*(*succ*(*x*),*0*), *Inr-fm*(*0*,*succ*(*Z*))))

lemma *is-Forall-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *Forall-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *Forall-fm-def*)

lemma *sats-Forall-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *Forall-fm*(*x*,*y*), *env*) ⟷
is-Forall(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *Forall-fm-def is-Forall-def*)

lemma *Forall-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-Forall*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *Forall-fm*(*i*,*j*), *env*)
by (*simp add*: *sats-Forall-fm*)

6.15.6 The Operator *is-and*, Internalized

definition
and-fm :: [*i*,*i*,*i*]=>*i* **where**
and-fm(*a*,*b*,*z*) ==
Or(*And*(*number1-fm*(*a*), *Equal*(*z*,*b*)),
And(*Neg*(*number1-fm*(*a*)),*empty-fm*(*z*)))

lemma *is-and-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *and-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *and-fm-def*)

lemma *sats-and-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *and-fm*(*x*,*y*,*z*), *env*) ⟷
is-and(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *and-fm-def is-and-def*)

lemma *is-and-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-and*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *and-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.15.7 The Operator *is-or*, Internalized

definition
  or-fm :: [*i*,*i*,*i*]=>*i* **where**
    or-fm(*a*,*b*,*z*) ==
      Or(*And*(*number1-fm*(*a*), *number1-fm*(*z*)),
        And(*Neg*(*number1-fm*(*a*)), *Equal*(*z*,*b*)))

lemma *is-or-type* [*TC*]:
    [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *or-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *or-fm-def*)

lemma *sats-or-fm* [*simp*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *or-fm*(*x*,*y*,*z*), *env*) ⟷
      is-or(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *or-fm-def is-or-def*)

lemma *is-or-iff-sats*:
    [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
      i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
    ==> *is-or*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *or-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.15.8 The Operator *is-not*, Internalized

definition
  not-fm :: [*i*,*i*]=>*i* **where**
    not-fm(*a*,*z*) ==
      Or(*And*(*number1-fm*(*a*), *empty-fm*(*z*)),
        And(*Neg*(*number1-fm*(*a*)), *number1-fm*(*z*)))

lemma *is-not-type* [*TC*]:
    [| *x* ∈ *nat*; *z* ∈ *nat* |] ==> *not-fm*(*x*,*z*) ∈ *formula*
by (*simp add*: *not-fm-def*)

lemma *sats-is-not-fm* [*simp*]:
  [| *x* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
  ==> *sats*(*A*, *not-fm*(*x*,*z*), *env*) ⟷ *is-not*(##*A*, *nth*(*x*,*env*), *nth*(*z*,*env*))
by (*simp add*: *not-fm-def is-not-def*)

lemma *is-not-iff-sats*:
    [| *nth*(*i*,*env*) = *x*; *nth*(*k*,*env*) = *z*;
      i ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
    ==> *is-not*(##*A*, *x*, *z*) ⟷ *sats*(*A*, *not-fm*(*i*,*k*), *env*)
by *simp*

6.16 Well-Founded Recursion!

6.16.1 The Operator *M-is-recfun*

Alternative definition, minimizing nesting of quantifiers around MH

```
lemma M-is-recfun-iff:
  M-is-recfun(M,MH,r,a,f) ⟷
   (∀z[M]. z ∈ f ⟷
    (∃x[M]. ∃f-r-sx[M]. ∃y[M].
           MH(x, f-r-sx, y) & pair(M,x,y,z) &
           (∃xa[M]. ∃sx[M]. ∃r-sx[M].
              pair(M,x,a,xa) & upair(M,x,x,sx) &
             pre-image(M,r,sx,r-sx) & restriction(M,f,r-sx,f-r-sx) &
             xa ∈ r)))
apply (simp add: M-is-recfun-def)
apply (rule rall-cong, blast)
done
```

The three arguments of p are always 2, 1, 0 and z

```
definition
  is-recfun-fm :: [i, i, i, i]=>i where
  is-recfun-fm(p,r,a,f) ==
   Forall(Iff(Member(0,succ(f)),
    Exists(Exists(Exists(
     And(p,
      And(pair-fm(2,0,3),
       Exists(Exists(Exists(
        And(pair-fm(5,a#+7,2),
         And(upair-fm(5,5,1),
          And(pre-image-fm(r#+7,1,0),
           And(restriction-fm(f#+7,0,4), Member(2,r#+7)))))))))))))))

lemma is-recfun-type [TC]:
     [| p ∈ formula; x ∈ nat; y ∈ nat; z ∈ nat |]
      ==> is-recfun-fm(p,x,y,z) ∈ formula
by (simp add: is-recfun-fm-def)


lemma sats-is-recfun-fm:
  assumes MH-iff-sats:
     !!a0 a1 a2 a3.
        [|a0∈A; a1∈A; a2∈A; a3∈A|]
     ==> MH(a2, a1, a0) ⟷ sats(A, p, Cons(a0,Cons(a1,Cons(a2,Cons(a3,env)))))
  shows
     [|x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
      ==> sats(A, is-recfun-fm(p,x,y,z), env) ⟷
          M-is-recfun(##A, MH, nth(x,env), nth(y,env), nth(z,env))
by (simp add: is-recfun-fm-def M-is-recfun-iff MH-iff-sats [THEN iff-sym])
```

```
lemma is-recfun-iff-sats:
  assumes MH-iff-sats:
     !!a0 a1 a2 a3.
       [|a0∈A; a1∈A; a2∈A; a3∈A|]
     ==> MH(a2, a1, a0) ⟷ sats(A, p, Cons(a0,Cons(a1,Cons(a2,Cons(a3,env)))))
  shows
  [| nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
     i ∈ nat; j ∈ nat; k ∈ nat; env ∈ list(A)|]
   ==> M-is-recfun(##A, MH, x, y, z) ⟷ sats(A, is-recfun-fm(p,i,j,k), env)
by (simp add: sats-is-recfun-fm [OF MH-iff-sats])
```

The additional variable in the premise, namely f', is essential. It lets *MH* depend upon *x*, which seems often necessary. The same thing occurs in *is-wfrec-reflection*.

6.16.2 The Operator *is-wfrec*

The three arguments of *p* are always 2, 1, 0; *p* is enclosed by 5 quantifiers.

```
definition
  is-wfrec-fm :: [i, i, i, i]=>i where
  is-wfrec-fm(p,r,a,z) ==
    Exists(And(is-recfun-fm(p, succ(r), succ(a), 0),
          Exists(Exists(Exists(Exists(
           And(Equal(2,a#+5), And(Equal(1,4), And(Equal(0,z#+5), p)))))))))
```

We call *p* with arguments a, f, z by equating them with the corresponding quantified variables with de Bruijn indices 2, 1, 0.

There's an additional existential quantifier to ensure that the environments in both calls to MH have the same length.

```
lemma is-wfrec-type [TC]:
     [| p ∈ formula; x ∈ nat; y ∈ nat; z ∈ nat |]
      ==> is-wfrec-fm(p,x,y,z) ∈ formula
by (simp add: is-wfrec-fm-def)

lemma sats-is-wfrec-fm:
  assumes MH-iff-sats:
     !!a0 a1 a2 a3 a4.
       [|a0∈A; a1∈A; a2∈A; a3∈A; a4∈A|]
     ==> MH(a2, a1, a0) ⟷ sats(A, p, Cons(a0,Cons(a1,Cons(a2,Cons(a3,Cons(a4,env))))))
  shows
     [|x ∈ nat; y < length(env); z < length(env); env ∈ list(A)|]
      ==> sats(A, is-wfrec-fm(p,x,y,z), env) ⟷
          is-wfrec(##A, MH, nth(x,env), nth(y,env), nth(z,env))
apply (frule-tac x=z in lt-length-in-nat, assumption)
apply (frule lt-length-in-nat, assumption)
apply (simp add: is-wfrec-fm-def sats-is-recfun-fm is-wfrec-def MH-iff-sats [THEN
iff-sym], blast)
```

done

lemma *is-wfrec-iff-sats*:
  assumes *MH-iff-sats*:
    !!*a0 a1 a2 a3 a4*.
      [|*a0*∈*A*; *a1*∈*A*; *a2*∈*A*; *a3*∈*A*; *a4*∈*A*|]
    ==> *MH*(*a2*, *a1*, *a0*) ⟷ *sats*(*A*, *p*, *Cons*(*a0*,*Cons*(*a1*,*Cons*(*a2*,*Cons*(*a3*,*Cons*(*a4*,*env*))))))
  shows
  [|*nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
    i ∈ *nat*; *j* < *length*(*env*); *k* < *length*(*env*); *env* ∈ *list*(*A*)|]
   ==> *is-wfrec*(##*A*, *MH*, *x*, *y*, *z*) ⟷ *sats*(*A*, *is-wfrec-fm*(*p*,*i*,*j*,*k*), *env*)
by (*simp add*: *sats-is-wfrec-fm* [*OF MH-iff-sats*])

6.17 For Datatypes

6.17.1 Binary Products, Internalized

definition
  cartprod-fm :: [*i*,*i*,*i*]=>*i* **where**

   cartprod-fm(*A*,*B*,*z*) ==
     Forall(*Iff*(*Member*(*0*,*succ*(*z*)),
            Exists(*And*(*Member*(*0*,*succ*(*succ*(*A*))),
                 Exists(*And*(*Member*(*0*,*succ*(*succ*(*succ*(*B*)))),
                      pair-fm(*1*,*0*,*2*)))))))

lemma *cartprod-type* [*TC*]:
   [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *cartprod-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *cartprod-fm-def*)

lemma *sats-cartprod-fm* [*simp*]:
  [| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
   ==> *sats*(*A*, *cartprod-fm*(*x*,*y*,*z*), *env*) ⟷
     cartprod(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *cartprod-fm-def cartprod-def*)

lemma *cartprod-iff-sats*:
   [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
     i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
   ==> *cartprod*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *cartprod-fm*(*i*,*j*,*k*), *env*)
by (*simp add*: *sats-cartprod-fm*)

6.17.2 Binary Sums, Internalized

definition
  sum-fm :: [*i*,*i*,*i*]=>*i* **where**
   sum-fm(*A*,*B*,*Z*) ==
     Exists(*Exists*(*Exists*(*Exists*(
      And(*number1-fm*(*2*),

And(cartprod-fm(2,A#+4,3),
And(upair-fm(2,2,1),
And(cartprod-fm(1,B#+4,0), union-fm(3,0,Z#+4))))))))

lemma *sum-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *sum-fm(x,y,z)* ∈ *formula*
by (*simp add*: *sum-fm-def*)

lemma *sats-sum-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *sum-fm(x,y,z)*, *env*) ⟷
is-sum(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *sum-fm-def is-sum-def*)

lemma *sum-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-sum*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *sum-fm(i,j,k)*, *env*)
by *simp*

6.17.3 The Operator *quasinat*

definition
quasinat-fm :: *i=>i* **where**
quasinat-fm(z) == *Or*(*empty-fm(z)*, *Exists*(*succ-fm*(*0*,*succ*(*z*))))

lemma *quasinat-type* [*TC*]:
x ∈ *nat* ==> *quasinat-fm(x)* ∈ *formula*
by (*simp add*: *quasinat-fm-def*)

lemma *sats-quasinat-fm* [*simp*]:
[| *x* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *quasinat-fm(x)*, *env*) ⟷ *is-quasinat*(##*A*, *nth*(*x*,*env*))
by (*simp add*: *quasinat-fm-def is-quasinat-def*)

lemma *quasinat-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-quasinat*(##*A*, *x*) ⟷ *sats*(*A*, *quasinat-fm(i)*, *env*)
by *simp*

6.17.4 The Operator *is-nat-case*

I could not get it to work with the more natural assumption that *is-b* takes two arguments. Instead it must be a formula where 1 and 0 stand for *m* and *b*, respectively.

The formula *is-b* has free variables 1 and 0.

definition

```
  is-nat-case-fm :: [i, i, i, i]=>i where
 is-nat-case-fm(a,is-b,k,z) ==
    And(Implies(empty-fm(k), Equal(z,a)),
        And(Forall(Implies(succ-fm(0,succ(k)),
                   Forall(Implies(Equal(0,succ(succ(z))), is-b)))),
            Or(quasinat-fm(k), empty-fm(z))))

lemma is-nat-case-type [TC]:
     [| is-b ∈ formula;
         x ∈ nat; y ∈ nat; z ∈ nat |]
      ==> is-nat-case-fm(x,is-b,y,z) ∈ formula
by (simp add: is-nat-case-fm-def)

lemma sats-is-nat-case-fm:
  assumes is-b-iff-sats:
     !!a. a ∈ A ==> is-b(a,nth(z, env)) ⟷
                   sats(A, p, Cons(nth(z,env), Cons(a, env)))
  shows
     [|x ∈ nat; y ∈ nat; z < length(env); env ∈ list(A)|]
      ==> sats(A, is-nat-case-fm(x,p,y,z), env) ⟷
          is-nat-case(##A, nth(x,env), is-b, nth(y,env), nth(z,env))
apply (frule lt-length-in-nat, assumption)
apply (simp add: is-nat-case-fm-def is-nat-case-def is-b-iff-sats [THEN iff-sym])
done

lemma is-nat-case-iff-sats:
  [| (!!a. a ∈ A ==> is-b(a,z) ⟷
                   sats(A, p, Cons(z, Cons(a,env))));
     nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
     i ∈ nat; j ∈ nat; k < length(env); env ∈ list(A)|]
   ==> is-nat-case(##A, x, is-b, y, z) ⟷ sats(A, is-nat-case-fm(i,p,j,k), env)
by (simp add: sats-is-nat-case-fm [of A is-b])
```

The second argument of *is-b* gives it direct access to x, which is essential for handling free variable references. Without this argument, we cannot prove reflection for *iterates-MH*.

6.18 The Operator *iterates-MH*, Needed for Iteration

```
definition
  iterates-MH-fm :: [i, i, i, i, i]=>i where
 iterates-MH-fm(isF,v,n,g,z) ==
    is-nat-case-fm(v,
      Exists(And(fun-apply-fm(succ(succ(succ(g))),2,0),
                 Forall(Implies(Equal(0,2), isF)))),
      n, z)

lemma iterates-MH-type [TC]:
     [| p ∈ formula;
```

```
    v ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat |]
  ==> iterates-MH-fm(p,v,x,y,z) ∈ formula
by (simp add: iterates-MH-fm-def)

lemma sats-iterates-MH-fm:
 assumes is-F-iff-sats:
   !!a b c d. [| a ∈ A; b ∈ A; c ∈ A; d ∈ A|]
        ==> is-F(a,b) ⟷
          sats(A, p, Cons(b, Cons(a, Cons(c, Cons(d,env)))))
 shows
   [|v ∈ nat; x ∈ nat; y ∈ nat; z < length(env); env ∈ list(A)|]
   ==> sats(A, iterates-MH-fm(p,v,x,y,z), env) ⟷
      iterates-MH(##A, is-F, nth(v,env), nth(x,env), nth(y,env), nth(z,env))
apply (frule lt-length-in-nat, assumption)
apply (simp add: iterates-MH-fm-def iterates-MH-def sats-is-nat-case-fm
        is-F-iff-sats [symmetric])
apply (rule is-nat-case-cong)
apply (simp-all add: setclass-def)
done

lemma iterates-MH-iff-sats:
 assumes is-F-iff-sats:
   !!a b c d. [| a ∈ A; b ∈ A; c ∈ A; d ∈ A|]
        ==> is-F(a,b) ⟷
          sats(A, p, Cons(b, Cons(a, Cons(c, Cons(d,env)))))
 shows
 [| nth(i',env) = v; nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
    i' ∈ nat; i ∈ nat; j ∈ nat; k < length(env); env ∈ list(A)|]
  ==> iterates-MH(##A, is-F, v, x, y, z) ⟷
     sats(A, iterates-MH-fm(p,i',i,j,k), env)
by (simp add: sats-iterates-MH-fm [OF is-F-iff-sats])
```

The second argument of p gives it direct access to x, which is essential for handling free variable references. Without this argument, we cannot prove reflection for *list-N*.

6.18.1 The Operator *is-iterates*

The three arguments of p are always 2, 1, 0; p is enclosed by 9 (??) quantifiers.

```
definition
 is-iterates-fm :: [i, i, i, i]=>i where
 is-iterates-fm(p,v,n,Z) ==
   Exists(Exists(
    And(succ-fm(n#+2,1),
     And(Memrel-fm(1,0),
         is-wfrec-fm(iterates-MH-fm(p, v#+7, 2, 1, 0),
                 0, n#+2, Z#+2)))))
```

We call p with arguments a, f, z by equating them with the corresponding quantified variables with de Bruijn indices 2, 1, 0.

lemma *is-iterates-type* [*TC*]:
  [| *p* ∈ *formula*; *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |]
   ==> *is-iterates-fm*(*p*,*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *is-iterates-fm-def*)

lemma *sats-is-iterates-fm*:
 assumes *is-F-iff-sats*:
  !!*a b c d e f g h i j k*.
   [| *a* ∈ *A*; *b* ∈ *A*; *c* ∈ *A*; *d* ∈ *A*; *e* ∈ *A*; *f* ∈ *A*;
    g ∈ *A*; *h* ∈ *A*; *i* ∈ *A*; *j* ∈ *A*; *k* ∈ *A*|]
   ==> *is-F*(*a*,*b*) ⟷
    sats(*A*, *p*, *Cons*(*b*, *Cons*(*a*, *Cons*(*c*, *Cons*(*d*, *Cons*(*e*, *Cons*(*f*,
     Cons(*g*, *Cons*(*h*, *Cons*(*i*, *Cons*(*j*, *Cons*(*k*, *env*))))))))))))
 shows
  [|*x* ∈ *nat*; *y* < *length*(*env*); *z* < *length*(*env*); *env* ∈ *list*(*A*)|]
   ==> *sats*(*A*, *is-iterates-fm*(*p*,*x*,*y*,*z*), *env*) ⟷
    is-iterates(##*A*, *is-F*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
apply (*frule-tac x*=*z* **in** *lt-length-in-nat*, *assumption*)
apply (*frule lt-length-in-nat*, *assumption*)
apply (*simp add*: *is-iterates-fm-def is-iterates-def sats-is-nat-case-fm*
   is-F-iff-sats [*symmetric*] *sats-is-wfrec-fm sats-iterates-MH-fm*)
done

lemma *is-iterates-iff-sats*:
 assumes *is-F-iff-sats*:
  !!*a b c d e f g h i j k*.
   [| *a* ∈ *A*; *b* ∈ *A*; *c* ∈ *A*; *d* ∈ *A*; *e* ∈ *A*; *f* ∈ *A*;
    g ∈ *A*; *h* ∈ *A*; *i* ∈ *A*; *j* ∈ *A*; *k* ∈ *A*|]
   ==> *is-F*(*a*,*b*) ⟷
    sats(*A*, *p*, *Cons*(*b*, *Cons*(*a*, *Cons*(*c*, *Cons*(*d*, *Cons*(*e*, *Cons*(*f*,
     Cons(*g*, *Cons*(*h*, *Cons*(*i*, *Cons*(*j*, *Cons*(*k*, *env*))))))))))))
 shows
 [| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
   i ∈ *nat*; *j* < *length*(*env*); *k* < *length*(*env*); *env* ∈ *list*(*A*)|]
  ==> *is-iterates*(##*A*, *is-F*, *x*, *y*, *z*) ⟷
   sats(*A*, *is-iterates-fm*(*p*,*i*,*j*,*k*), *env*)
by (*simp add*: *sats-is-iterates-fm* [*OF is-F-iff-sats*])

The second argument of p gives it direct access to x, which is essential for handling free variable references. Without this argument, we cannot prove reflection for *list-N*.

6.18.2 The Formula *is-eclose-n*, Internalized

definition
 eclose-n-fm :: [*i*,*i*,*i*]=>*i* **where**

eclose-n-fm(*A*,*n*,*Z*) == *is-iterates-fm*(*big-union-fm*(*1*,*0*), *A*, *n*, *Z*)

lemma *eclose-n-fm-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *eclose-n-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *eclose-n-fm-def*)

lemma *sats-eclose-n-fm* [*simp*]:
[| *x* ∈ *nat*; *y* < *length*(*env*); *z* < *length*(*env*); *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *eclose-n-fm*(*x*,*y*,*z*), *env*) ⟷
is-eclose-n(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
apply (*frule-tac x*=*z* **in** *lt-length-in-nat*, *assumption*)
apply (*frule-tac x*=*y* **in** *lt-length-in-nat*, *assumption*)
apply (*simp add*: *eclose-n-fm-def is-eclose-n-def*
sats-is-iterates-fm)
done

lemma *eclose-n-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* < *length*(*env*); *k* < *length*(*env*); *env* ∈ *list*(*A*)|]
==> *is-eclose-n*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *eclose-n-fm*(*i*,*j*,*k*), *env*)
by (*simp add*: *sats-eclose-n-fm*)

6.18.3 Membership in *eclose*(*A*)

definition
mem-eclose-fm :: [*i*,*i*]=>*i* **where**
mem-eclose-fm(*x*,*y*) ==
Exists(*Exists*(
And(*finite-ordinal-fm*(*1*),
And(*eclose-n-fm*(*x*#+*2*,*1*,*0*), *Member*(*y*#+*2*,*0*)))))

lemma *mem-eclose-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *mem-eclose-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *mem-eclose-fm-def*)

lemma *sats-mem-eclose-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *mem-eclose-fm*(*x*,*y*), *env*) ⟷ *mem-eclose*(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *mem-eclose-fm-def mem-eclose-def*)

lemma *mem-eclose-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *mem-eclose*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *mem-eclose-fm*(*i*,*j*), *env*)
by *simp*

6.18.4 The Predicate "Is *eclose*(*A*)"

definition

is-eclose-fm :: [*i*,*i*]=>*i* **where**
is-eclose-fm(*A*,*Z*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*Z*)), *mem-eclose-fm*(*succ*(*A*),*0*)))

lemma *is-eclose-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *is-eclose-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *is-eclose-fm-def*)

lemma *sats-is-eclose-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *is-eclose-fm*(*x*,*y*), *env*) ⟷ *is-eclose*(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *is-eclose-fm-def is-eclose-def*)

lemma *is-eclose-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-eclose*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *is-eclose-fm*(*i*,*j*), *env*)
by *simp*

6.18.5 The List Functor, Internalized

definition
list-functor-fm :: [*i*,*i*,*i*]=>*i* **where**

list-functor-fm(*A*,*X*,*Z*) ==
Exists(*Exists*(
And(*number1-fm*(*1*),
And(*cartprod-fm*(*A*#+*2*,*X*#+*2*,*0*), *sum-fm*(*1*,*0*,*Z*#+*2*)))))

lemma *list-functor-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *list-functor-fm*(*x*,*y*,*z*) ∈ *formula*
by (*simp add*: *list-functor-fm-def*)

lemma *sats-list-functor-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *list-functor-fm*(*x*,*y*,*z*), *env*) ⟷
is-list-functor(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*), *nth*(*z*,*env*))
by (*simp add*: *list-functor-fm-def is-list-functor-def*)

lemma *list-functor-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*; *nth*(*k*,*env*) = *z*;
i ∈ *nat*; *j* ∈ *nat*; *k* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-list-functor*(##*A*, *x*, *y*, *z*) ⟷ *sats*(*A*, *list-functor-fm*(*i*,*j*,*k*), *env*)
by *simp*

6.18.6 The Formula *is-list-N*, Internalized

definition
list-N-fm :: [*i*,*i*,*i*]=>*i* **where**
list-N-fm(*A*,*n*,*Z*) ==

```
    Exists(
      And(empty-fm(0),
          is-iterates-fm(list-functor-fm(A#+9#+3,1,0), 0, n#+1, Z#+1)))
```

lemma *list-N-fm-type* [*TC*]:
 [| x ∈ nat; y ∈ nat; z ∈ nat |] ==> list-N-fm(x,y,z) ∈ formula
by (*simp add*: *list-N-fm-def*)

lemma *sats-list-N-fm* [*simp*]:
```
  [| x ∈ nat; y < length(env); z < length(env); env ∈ list(A)|]
   ==> sats(A, list-N-fm(x,y,z), env) ⟷
       is-list-N(##A, nth(x,env), nth(y,env), nth(z,env))
```
apply (*frule-tac x=z* **in** *lt-length-in-nat*, *assumption*)
apply (*frule-tac x=y* **in** *lt-length-in-nat*, *assumption*)
apply (*simp add*: *list-N-fm-def is-list-N-def sats-is-iterates-fm*)
done

lemma *list-N-iff-sats*:
```
    [| nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
       i ∈ nat; j < length(env); k < length(env); env ∈ list(A)|]
    ==> is-list-N(##A, x, y, z) ⟷ sats(A, list-N-fm(i,j,k), env)
```
by (*simp add*: *sats-list-N-fm*)

6.18.7 The Predicate "Is A List"

definition
```
  mem-list-fm :: [i,i]=>i where
    mem-list-fm(x,y) ==
      Exists(Exists(
        And(finite-ordinal-fm(1),
          And(list-N-fm(x#+2,1,0), Member(y#+2,0)))))
```

lemma *mem-list-type* [*TC*]:
    [| x ∈ nat; y ∈ nat |] ==> mem-list-fm(x,y) ∈ formula
by (*simp add*: *mem-list-fm-def*)

lemma *sats-mem-list-fm* [*simp*]:
```
  [| x ∈ nat; y ∈ nat; env ∈ list(A)|]
  ==> sats(A, mem-list-fm(x,y), env) ⟷ mem-list(##A, nth(x,env), nth(y,env))
```
by (*simp add*: *mem-list-fm-def mem-list-def*)

lemma *mem-list-iff-sats*:
```
    [| nth(i,env) = x; nth(j,env) = y;
       i ∈ nat; j ∈ nat; env ∈ list(A)|]
    ==> mem-list(##A, x, y) ⟷ sats(A, mem-list-fm(i,j), env)
```
by *simp*

6.18.8 The Predicate "Is *list*(*A*)"

definition

is-list-fm :: [*i*,*i*]=>*i* **where**
is-list-fm(*A*,*Z*) ==
Forall(*Iff*(*Member*(*0*,*succ*(*Z*)), *mem-list-fm*(*succ*(*A*),*0*)))

lemma *is-list-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *is-list-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *is-list-fm-def*)

lemma *sats-is-list-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *is-list-fm*(*x*,*y*), *env*) ⟷ *is-list*(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *is-list-fm-def is-list-def*)

lemma *is-list-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-list*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *is-list-fm*(*i*,*j*), *env*)
by *simp*

6.18.9 The Formula Functor, Internalized

definition *formula-functor-fm* :: [*i*,*i*]=>*i* **where**

formula-functor-fm(*X*,*Z*) ==
Exists(*Exists*(*Exists*(*Exists*(*Exists*(
And(*omega-fm*(*4*),
And(*cartprod-fm*(*4*,*4*,*3*),
And(*sum-fm*(*3*,*3*,*2*),
And(*cartprod-fm*(*X*#+*5*,*X*#+*5*,*1*),
And(*sum-fm*(*1*,*X*#+*5*,*0*), *sum-fm*(*2*,*0*,*Z*#+*5*))))))))))

lemma *formula-functor-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *formula-functor-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *formula-functor-fm-def*)

lemma *sats-formula-functor-fm* [*simp*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *formula-functor-fm*(*x*,*y*), *env*) ⟷
is-formula-functor(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
by (*simp add*: *formula-functor-fm-def is-formula-functor-def*)

lemma *formula-functor-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i ∈ *nat*; *j* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *is-formula-functor*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *formula-functor-fm*(*i*,*j*), *env*)
by *simp*

6.18.10 The Formula *is-formula-N*, Internalized

definition

formula-N-fm :: [*i*,*i*]=>*i* **where**
formula-N-fm(*n*,*Z*) ==
Exists(
And(*empty-fm*(*0*),
is-iterates-fm(*formula-functor-fm*(*1*,*0*), *0*, *n*#+*1*, *Z*#+*1*)))

lemma *formula-N-fm-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat* |] ==> *formula-N-fm*(*x*,*y*) ∈ *formula*
by (*simp add*: *formula-N-fm-def*)

lemma *sats-formula-N-fm* [*simp*]:
[| *x* < *length*(*env*); *y* < *length*(*env*); *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *formula-N-fm*(*x*,*y*), *env*) ⟷
is-formula-N(##*A*, *nth*(*x*,*env*), *nth*(*y*,*env*))
apply (*frule-tac x*=*y* **in** *lt-length-in-nat*, *assumption*)
apply (*frule lt-length-in-nat*, *assumption*)
apply (*simp add*: *formula-N-fm-def is-formula-N-def sats-is-iterates-fm*)
done

lemma *formula-N-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *nth*(*j*,*env*) = *y*;
i < *length*(*env*); *j* < *length*(*env*); *env* ∈ *list*(*A*)|]
==> *is-formula-N*(##*A*, *x*, *y*) ⟷ *sats*(*A*, *formula-N-fm*(*i*,*j*), *env*)
by (*simp add*: *sats-formula-N-fm*)

6.18.11 The Predicate "Is A Formula"

definition
mem-formula-fm :: *i*=>*i* **where**
mem-formula-fm(*x*) ==
Exists(*Exists*(
And(*finite-ordinal-fm*(*1*),
And(*formula-N-fm*(*1*,*0*), *Member*(*x*#+*2*,*0*)))))

lemma *mem-formula-type* [*TC*]:
x ∈ *nat* ==> *mem-formula-fm*(*x*) ∈ *formula*
by (*simp add*: *mem-formula-fm-def*)

lemma *sats-mem-formula-fm* [*simp*]:
[| *x* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *sats*(*A*, *mem-formula-fm*(*x*), *env*) ⟷ *mem-formula*(##*A*, *nth*(*x*,*env*))
by (*simp add*: *mem-formula-fm-def mem-formula-def*)

lemma *mem-formula-iff-sats*:
[| *nth*(*i*,*env*) = *x*; *i* ∈ *nat*; *env* ∈ *list*(*A*)|]
==> *mem-formula*(##*A*, *x*) ⟷ *sats*(*A*, *mem-formula-fm*(*i*), *env*)
by *simp*

6.18.12 The Predicate "Is *formula*"

definition
is-formula-fm :: *i*=>*i* **where**
is-formula-fm(*Z*) == *Forall*(*Iff*(*Member*(*0*,*succ*(*Z*)), *mem-formula-fm*(*0*)))

lemma *is-formula-type* [*TC*]:
$x \in nat$ ==> *is-formula-fm*(*x*) $\in$ *formula*
by (*simp add*: *is-formula-fm-def*)

lemma *sats-is-formula-fm* [*simp*]:
[| $x \in nat$; $env \in list(A)$|]
==> *sats*(*A*, *is-formula-fm*(*x*), *env*) $\longleftrightarrow$ *is-formula*(##*A*, *nth*(*x*,*env*))
by (*simp add*: *is-formula-fm-def is-formula-def*)

lemma *is-formula-iff-sats*:
[| *nth*(*i*,*env*) = *x*; $i \in nat$; $env \in list(A)$|]
==> *is-formula*(##*A*, *x*) $\longleftrightarrow$ *sats*(*A*, *is-formula-fm*(*i*), *env*)
by *simp*

6.18.13 The Operator *is-transrec*

The three arguments of p are always 2, 1, 0. It is buried within eight quantifiers! We call p with arguments a, f, z by equating them with the corresponding quantified variables with de Bruijn indices 2, 1, 0.

definition
is-transrec-fm :: [*i*, *i*, *i*]=>*i* **where**
is-transrec-fm(*p*,*a*,*z*) ==
Exists(*Exists*(*Exists*(
And(*upair-fm*(*a*#+*3*,*a*#+*3*,*2*),
And(*is-eclose-fm*(*2*,*1*),
And(*Memrel-fm*(*1*,*0*), *is-wfrec-fm*(*p*,*0*,*a*#+*3*,*z*#+*3*)))))))

lemma *is-transrec-type* [*TC*]:
[| $p \in formula$; $x \in nat$; $z \in nat$ |]
==> *is-transrec-fm*(*p*,*x*,*z*) $\in$ *formula*
by (*simp add*: *is-transrec-fm-def*)

lemma *sats-is-transrec-fm*:
assumes *MH-iff-sats*:
!!*a0 a1 a2 a3 a4 a5 a6 a7*.
[|*a0*$\in$*A*; *a1*$\in$*A*; *a2*$\in$*A*; *a3*$\in$*A*; *a4*$\in$*A*; *a5*$\in$*A*; *a6*$\in$*A*; *a7*$\in$*A*|]
==> *MH*(*a2*, *a1*, *a0*) $\longleftrightarrow$
sats(*A*, *p*, *Cons*(*a0*,*Cons*(*a1*,*Cons*(*a2*,*Cons*(*a3*,
Cons(*a4*,*Cons*(*a5*,*Cons*(*a6*,*Cons*(*a7*,*env*)))))))))
shows
[|*x* < *length*(*env*); *z* < *length*(*env*); $env \in list(A)$|]
==> *sats*(*A*, *is-transrec-fm*(*p*,*x*,*z*), *env*) $\longleftrightarrow$

is-transrec(##*A*, *MH*, *nth*(*x*,*env*), *nth*(*z*,*env*))
apply (*frule-tac x*=*z* **in** *lt-length-in-nat*, *assumption*)
apply (*frule-tac x*=*x* **in** *lt-length-in-nat*, *assumption*)
apply (*simp add*: *is-transrec-fm-def sats-is-wfrec-fm is-transrec-def MH-iff-sats* [*THEN iff-sym*])
done

lemma *is-transrec-iff-sats*:
assumes *MH-iff-sats*:
!!*a0 a1 a2 a3 a4 a5 a6 a7*.
[|*a0*∈*A*; *a1*∈*A*; *a2*∈*A*; *a3*∈*A*; *a4*∈*A*; *a5*∈*A*; *a6*∈*A*; *a7*∈*A*|]
==> *MH*(*a2*, *a1*, *a0*) ⟷
sats(*A*, *p*, *Cons*(*a0*,*Cons*(*a1*,*Cons*(*a2*,*Cons*(*a3*,
Cons(*a4*,*Cons*(*a5*,*Cons*(*a6*,*Cons*(*a7*,*env*)))))))))
shows
[|*nth*(*i*,*env*) = *x*; *nth*(*k*,*env*) = *z*;
i < *length*(*env*); *k* < *length*(*env*); *env* ∈ *list*(*A*)|]
==> *is-transrec*(##*A*, *MH*, *x*, *z*) ⟷ *sats*(*A*, *is-transrec-fm*(*p*,*i*,*k*), *env*)
by (*simp add*: *sats-is-transrec-fm* [*OF MH-iff-sats*])

end
theory *Nat-Miscellanea* **imports** *ZF* **begin**

7 Auxiliary results

lemmas *nat-succI* = *Ord-succ-mem-iff* [*THEN iffD2*,*OF nat-into-Ord*]

lemma *nat-succD* : *m* ∈ *nat* ⟹ *succ*(*n*) ∈ *succ*(*m*) ⟹ *n* ∈ *m*
by (*drule-tac j*=*succ*(*m*) **in** *ltI*,*auto elim*:*ltD*)

lemmas *zero-in* = *ltD* [*OF nat-0-le*]

lemma *in-n-in-nat* : *m* ∈ *nat* ⟹ *n* ∈ *m* ⟹ *n* ∈ *nat*
by(*drule ltI*[*of n*],*auto simp add*: *lt-nat-in-nat*)

lemma *in-succ-in-nat* : *m* ∈ *nat* ⟹ *n* ∈ *succ*(*m*) ⟹ *n* ∈ *nat*
by(*auto simp add*:*in-n-in-nat*)

lemma *ltI-neg* : *x* ∈ *nat* ⟹ *j* ≤ *x* ⟹ *j* ≠ *x* ⟹ *j* < *x*
by (*simp add*: *le-iff*)

lemma *succ-pred-eq* : *m* ∈ *nat* ⟹ *m* ≠ *0* ⟹ *succ*(*pred*(*m*)) = *m*
by (*auto elim*: *natE*)

lemma *succ-ltI* : *n* ∈ *nat* ⟹ *succ*(*j*) < *n* ⟹ *j* < *n*
apply (*rule-tac j*=*succ*(*j*) **in** *lt-trans*,*rule le-refl*,*rule Ord-succD*)
apply (*rule nat-into-Ord*,*erule in-n-in-nat*,*erule ltD*,*simp*)

done

lemma *succ-In* : $n \in nat \Longrightarrow succ(j) \in n \Longrightarrow j \in n$
by (*rule succ-ltI*[*THEN ltD*], *auto intro*: *ltI*)

lemmas *succ-leD* = *succ-leE*[*OF leI*]

lemma *succpred-leI* : $n \in nat \Longrightarrow n \leq succ(pred(n))$
by (*auto elim*: *natE*)

lemma *succpred-n0* : $p \in nat \Longrightarrow succ(n) \in p \Longrightarrow p \neq 0$
by (*auto elim*: *natE*)

lemma *funcI* : $f \in A \to B \Longrightarrow a \in A \Longrightarrow b = f \text{ ` } a \Longrightarrow \langle a, b\rangle \in f$
by(*simp-all add*: *apply-Pair*)

lemmas *natEin* = *natE* [*OF lt-nat-in-nat*]

lemma *succ-in* : $succ(x) \leq y \Longrightarrow x \in y$
by (*auto dest*:*ltD*)

lemmas *Un-least-lt-iffn* = *Un-least-lt-iff* [*OF nat-into-Ord nat-into-Ord*]

lemma *pred-le2* : $n \in nat \Longrightarrow m \in nat \Longrightarrow pred(n) \leq m \Longrightarrow n \leq succ(m)$
by(*subgoal-tac n*∈*nat*,*rule-tac n*=*n* **in** *natE*,*auto*)

lemma *pred-le* : $n \in nat \Longrightarrow m \in nat \Longrightarrow n \leq succ(m) \Longrightarrow pred(n) \leq m$
by(*subgoal-tac pred*(*n*)∈*nat*,*rule-tac n*=*n* **in** *natE*,*auto*)

lemma *un-leD1* : $i \in nat \Longrightarrow j \in nat \Longrightarrow k \in nat \Longrightarrow i \cup j \leq k \Longrightarrow i \leq k$
by (*rule Un-least-lt-iff*[*THEN iffD1*[*THEN conjunct1*]],*simp-all*)

lemma *un-leD2* : $i \in nat \Longrightarrow j \in nat \Longrightarrow k \in nat \Longrightarrow i \cup j \leq k \Longrightarrow j \leq k$
by (*rule Un-least-lt-iff*[*THEN iffD1*[*THEN conjunct2*]],*simp-all*)

lemma *gt1* : $n \in nat \Longrightarrow i \in n \Longrightarrow i \neq 0 \Longrightarrow i \neq 1 \Longrightarrow 1 < i$
by(*rule-tac n*=*i* **in** *natE*,*erule in-n-in-nat*,*auto intro*: *Ord-0-lt*)

lemma *pred-mono* : $m \in nat \Longrightarrow n \leq m \Longrightarrow pred(n) \leq pred(m)$
by(*rule-tac n*=*n* **in** *natE*,*auto simp add*:*le-in-nat*,*erule-tac n*=*m* **in** *natE*,*auto*)

lemma *pred2-Un*:
assumes $j \in nat$ $m \leq j$ $n \leq j$
shows $pred(pred(m \cup n)) \leq pred(pred(j))$
using *assms pred-mono*[*of j*] *le-in-nat Un-least-lt pred-mono* **by** *simp*

lemma *nat-union-abs1* :
$[\![Ord(i) ; Ord(j) ; i \leq j]\!] \Longrightarrow i \cup j = j$

by (*rule Un-absorb1*,*erule le-imp-subset*)

lemma *nat-union-abs2* :
⟦ *Ord*(*i*) ; *Ord*(*j*) ; *i* ≤ *j* ⟧ ⟹ *j* ∪ *i* = *j*
by (*rule Un-absorb2*,*erule le-imp-subset*)

lemma *nat-un-max* : *i* ∈ *nat* ⟹ *j* ∈ *nat* ⟹ *i* ∪ *j* = *max*(*i*,*j*)
apply(*auto simp add*:*max-def nat-union-abs1*)
apply(*auto simp add*: *not-lt-iff-le leI nat-union-abs2*)
done

lemma *nat-un-ty* : *i*∈*nat* ⟹*j*∈*nat* ⟹ *i*∪*j* ∈ *nat*
by *simp*

lemma *nat-max-ty* : *i*∈*nat* ⟹*j*∈*nat* ⟹ *max*(*i*,*j*) ∈ *nat*
unfolding *max-def* **by** *simp*

lemmas *nat-simp-union* = *nat-un-max nat-un-ty nat-max-ty max-def*

end
theory *Renaming*
imports
Nat-Miscellanea
~~/src/ZF/Constructible/Formula
begin

8 Auxiliary results

lemma *app-nm* : *n*∈*nat* ⟹ *m*∈*nat* ⟹ *f*∈*n*→*m* ⟹ *x* ∈ *nat* ⟹ *f*‘*x* ∈ *nat*
apply(*case-tac x*∈*n*,*rule-tac m*=*m* **in** *in-n-in-nat*,(*simp add*:*apply-type*)+)
apply(*subst apply-0*,*subst domain-of-fun*,*assumption*+,*auto*)
done

9 Renaming of free variables

definition
sum-id :: [*i*,*i*] ⇒ *i* **where**
sum-id(*m*,*f*) == λ*j* ∈ *succ*(*m*) . *if j*=*0 then 0 else succ*(*f*‘*pred*(*j*))

lemma *sum-id0* : *sum-id*(*m*,*f*)‘*0* = *0*
by(*unfold sum-id-def*,*simp*)

lemma *sum-idS* : *succ*(*x*) ∈ *succ*(*m*) ⟹ *sum-id*(*m*,*f*)‘*succ*(*x*) = *succ*(*f*‘*x*)
by(*unfold sum-id-def*,*simp*)

lemma *sum-id-tc* :
n ∈ *nat* ⟹ *m* ∈ *nat* ⟹ *f* ∈ *n* → *m* ⟹ *sum-id*(*n*,*f*) ∈ *succ*(*n*) → *succ*(*m*)
apply (*rule Pi-iff* [*THEN iffD2*],*rule conjI*)

```
 apply (unfold sum-id-def,rule function-lam)
 apply (rule conjI,auto)
 apply (erule-tac p=x and A=succ(n) and
     b=λ i. if i = 0 then 0 else succ(f‘pred(i)) and
     P=x∈succ(n)×succ(m) in lamE)
 apply(rename-tac j,case-tac j=0,simp,simp add:zero-in)
 apply(subgoal-tac f‘pred(j) ∈ m,simp)
  apply(rule nat-succI,assumption+)
 apply (erule-tac A=n in apply-type)
 apply (rule Ord-succ-mem-iff [THEN iffD1],simp)
apply (subst succ-pred-eq,rule-tac A=succ(n) in subsetD,rule naturals-subset-nat)
   apply (simp+)
 done
```

10 Renaming of formulas

```
consts  ren :: i=>i
primrec
 ren(Member(x,y)) =
     (λ n ∈ nat . λ m ∈ nat. λf ∈ n → m. Member (f‘x, f‘y))

 ren(Equal(x,y)) =
     (λ n ∈ nat . λ m ∈ nat. λf ∈ n → m. Equal (f‘x, f‘y))

 ren(Nand(p,q)) =
     (λ n ∈ nat . λ m ∈ nat. λf ∈ n → m. Nand (ren(p)‘n‘m‘f, ren(q)‘n‘m‘f))

 ren(Forall(p)) =
    (λ n ∈ nat . λ m ∈ nat. λf ∈ n → m. Forall (ren(p)‘succ(n)‘succ(m)‘sum-id(n,f)))

lemma arity-meml : l ∈ nat =⇒ Member(x,y) ∈ formula =⇒ arity(Member(x,y))
≤ l =⇒ x ∈ l
 by(simp,rule subsetD,rule le-imp-subset,assumption,simp)
lemma arity-memr : l ∈ nat =⇒ Member(x,y) ∈ formula =⇒ arity(Member(x,y))
≤ l =⇒ y ∈ l
 by(simp,rule subsetD,rule le-imp-subset,assumption,simp)
lemma arity-eql : l ∈ nat =⇒ Equal(x,y) ∈ formula =⇒ arity(Equal(x,y)) ≤ l
=⇒ x ∈ l
 by(simp,rule subsetD,rule le-imp-subset,assumption,simp)
lemma arity-eqr : l ∈ nat =⇒ Equal(x,y) ∈ formula =⇒ arity(Equal(x,y)) ≤ l
=⇒ y ∈ l
 by(simp,rule subsetD,rule le-imp-subset,assumption,simp)
lemma  nand-ar1 : p ∈ formula =⇒ q∈formula =⇒arity(p) ≤ arity(Nand(p,q))
 by (simp,rule Un-upper1-le,simp+)
lemma nand-ar2 : p ∈ formula =⇒ q∈formula =⇒arity(q) ≤ arity(Nand(p,q))
 by (simp,rule Un-upper2-le,simp+)

lemma nand-ar1D : p ∈ formula =⇒ q∈formula =⇒ arity(Nand(p,q)) ≤ n =⇒
arity(p) ≤ n
```

by(*auto simp add*: *le-trans*[*OF Un-upper1-le*[*of arity*(*p*) *arity*(*q*)]])
lemma *nand-ar2D* : *p* ∈ *formula* ⟹ *q*∈*formula* ⟹ *arity*(*Nand*(*p*,*q*)) ≤ *n* ⟹ *arity*(*q*) ≤ *n*
by(*auto simp add*: *le-trans*[*OF Un-upper2-le*[*of arity*(*p*) *arity*(*q*)]])

lemma *ren-tc* : *p* ∈ *formula* ⟹
(⋀ *n m f* . *n* ∈ *nat* ⟹ *m* ∈ *nat* ⟹ *f* ∈ *n*→*m* ⟹ *ren*(*p*)‘*n*‘*m*‘*f* ∈ *formula*)
by (*induct set*:*formula*,*auto simp add*: *app-nm sum-id-tc*)

lemma *ren-arity* :
fixes *p*
assumes *p* ∈ *formula*
shows ⋀ *n m f* . *n* ∈ *nat* ⟹ *m* ∈ *nat* ⟹ *f* ∈ *n*→*m* ⟹ *arity*(*p*) ≤ *n* ⟹ *arity*(*ren*(*p*)‘*n*‘*m*‘*f*)≤*m*
using *assms*
proof (*induct set*:*formula*)
case (*Member x y*)
then have *f*‘*x* ∈ *m* *f*‘*y* ∈ *m*
using *Member assms* **by** (*simp add*: *arity-meml apply-funtype*,*simp add*:*arity-memr apply-funtype*)
then show *?case* **using** *Member* **by** (*simp add*: *Un-least-lt ltI*)
next
case (*Equal x y*)
then have *f*‘*x* ∈ *m* *f*‘*y* ∈ *m*
using *Equal assms* **by** (*simp add*: *arity-eql apply-funtype*,*simp add*:*arity-eqr apply-funtype*)
then show *?case* **using** *Equal* **by** (*simp add*: *Un-least-lt ltI*)
next
case (*Nand p q*)
then have *arity*(*p*)≤*arity*(*Nand*(*p*,*q*))
arity(*q*)≤*arity*(*Nand*(*p*,*q*))
by (*subst nand-ar1*,*simp*,*simp*,*simp*,*subst nand-ar2*,*simp*+)
then have *arity*(*p*)≤*n*
and *arity*(*q*)≤*n* **using** *Nand*
by (*rule-tac j*=*arity*(*Nand*(*p*,*q*)) **in** *le-trans*,*simp*,*simp*)+
then have *arity*(*ren*(*p*)‘*n*‘*m*‘*f*) ≤ *m* **and** *arity*(*ren*(*q*)‘*n*‘*m*‘*f*) ≤ *m*
using *Nand* **by** *auto*
then show *?case* **using** *Nand* **by** (*simp add*:*Un-least-lt*)
next
case (*Forall p*)
from *Forall* **have** *succ*(*n*)∈*nat* *succ*(*m*)∈*nat* **by** *auto*
from *Forall* **have** *2*: *sum-id*(*n*,*f*) ∈ *succ*(*n*)→*succ*(*m*) **by** (*simp add*:*sum-id-tc*)
from *Forall* **have** *3*:*arity*(*p*) ≤ *succ*(*n*) **by** (*rule-tac n*=*arity*(*p*) **in** *natE*,*simp*+)
then have *arity*(*ren*(*p*)‘*succ*(*n*)‘*succ*(*m*)‘*sum-id*(*n*,*f*))≤*succ*(*m*) **using** *Forall* ‹*succ*(*n*)∈*nat*› ‹*succ*(*m*)∈*nat*› *2* **by** *force*
then show *?case* **using** *Forall 2 3 ren-tc arity-type pred-le* **by** *auto*
qed

```
lemma forall-arityE : p ∈ formula ⟹ m ∈ nat ⟹ arity(Forall(p)) ≤ m ⟹
arity(p) ≤ succ(m)
  by(rule-tac n=arity(p) in natE,erule arity-type,simp+)

lemma env-coincidence-sum-id :
  assumes m ∈ nat n ∈ nat
    ϱ ∈ list(A) ϱ' ∈ list(A)
    f ∈ n → m
    ⋀ i . i < n ⟹ nth(i,ϱ) = nth(f`i,ϱ')
    a ∈ A j ∈ succ(n)
  shows nth(j,Cons(a,ϱ)) = nth(sum-id(n,f)`j,Cons(a,ϱ'))
proof −
  let ?g=sum-id(n,f)
  have succ(n) ∈ nat using ‹n∈nat› by simp
  then have j ∈ nat using ‹j∈succ(n)› in-n-in-nat by blast
  then have nth(j,Cons(a,ϱ)) = nth(?g`j,Cons(a,ϱ'))
  proof (cases rule:natE[OF ‹j∈nat›])
    case 1
    then show ?thesis using assms sum-id0 by simp
  next
    case (2 i)
    with ‹j∈succ(n)› have succ(i)∈succ(n) by simp
    with ‹n∈nat› have i ∈ n using nat-succD assms by simp
    have f`i∈m using ‹f∈n→m› apply-type ‹i∈n› by simp
    then have f`i ∈ nat using in-n-in-nat ‹m∈nat› by simp
    have nth(succ(i),Cons(a,ϱ)) = nth(i,ϱ) using ‹i∈nat› by simp
    also have ... = nth(f`i,ϱ') using assms ‹i∈n› ltI by simp
    also have ... = nth(succ(f`i),Cons(a,ϱ')) using ‹f`i∈nat› by simp
    also have ... = nth(?g`succ(i),Cons(a,ϱ'))
      using sum-idS ‹succ(i)∈succ(n)› cases by simp
    finally have nth(succ(i),Cons(a,ϱ)) = nth(?g`succ(i),Cons(a,ϱ')) .
    then show ?thesis using ‹j=succ(i)› by simp
  qed
  then show ?thesis .
qed

lemma sats-iff-sats-ren :
  fixes φ
  assumes φ ∈ formula
  shows ⟦ n ∈ nat ; m ∈ nat ; ϱ ∈ list(M) ; ϱ' ∈ list(M) ; f ∈ n → m ;
          arity(φ) ≤ n ;
          ⋀ i . i < n ⟹ nth(i,ϱ) = nth(f`i,ϱ') ⟧ ⟹
        sats(M,φ,ϱ) ⟷ sats(M,ren(φ)`n`m`f,ϱ')
  using ‹φ ∈ formula›
proof(induct φ arbitrary:n m ϱ ϱ' f)
  case (Member x y)
 have 0: ren(Member(x,y))`n`m`f = Member(f`x,f`y) using Member assms arity-type
by force
```

have *1*: $x \in n$ **using** *Member arity-meml* **by** *simp*
have $y \in n$ **using** *Member arity-memr* **by** *simp*
then show *?case* **using** *Member 1 0 ltI* **by** *simp*
next
case (*Equal x y*)
have *0*: $ren(Equal(x,y))`n`m`f = Equal(f`x,f`y)$ **using** *Equal assms arity-type* **by** *force*
have *1*: $x \in n$ **using** *Equal arity-eql* **by** *simp*
have $y \in n$ **using** *Equal arity-eqr* **by** *simp*
then show *?case* **using** *Equal 1 0 ltI* **by** *simp*
next
case (*Nand p q*)
have *0*:$ren(Nand(p,q))`n`m`f = Nand(ren(p)`n`m`f,ren(q)`n`m`f)$ **using** *Nand* **by** *simp*
have $arity(p) \leq n$ **using** *Nand nand-ar1D* **by** *simp*
then have *1*:$i \in arity(p) \Longrightarrow i \in n$ **for** *i* **using** *subsetD*[*OF le-imp-subset*[*OF* ‹$arity(p) \leq n$›]] **by** *simp*
then have $i \in arity(p) \Longrightarrow nth(i,\varrho) = nth(f`i,\varrho')$ **for** *i* **using** *Nand ltI* **by** *simp*
then have *2*:$sats(M,p,\varrho) \longleftrightarrow sats(M,ren(p)`n`m`f,\varrho')$ **using** ‹$arity(p)\leq n$› *1 Nand* **by** *simp*
have $arity(q) \leq n$ **using** *Nand nand-ar2D* **by** *simp*
then have *3*:$i \in arity(q) \Longrightarrow i \in n$ **for** *i* **using** *subsetD*[*OF le-imp-subset*[*OF* ‹$arity(q) \leq n$›]] **by** *simp*
then have $i \in arity(q) \Longrightarrow nth(i,\varrho) = nth(f`i,\varrho')$ **for** *i* **using** *Nand ltI* **by** *simp*
then have *4*:$sats(M,q,\varrho) \longleftrightarrow sats(M,ren(q)`n`m`f,\varrho')$ **using** *assms* ‹$arity(q)\leq n$› *3 Nand* **by** *simp*
then show *?case* **using** *Nand 0 2 4* **by** *simp*
next
case (*Forall p*)
have *0*:$ren(Forall(p))`n`m`f = Forall(ren(p)`succ(n)`succ(m)`sum\text{-}id(n,f))$
using *Forall* **by** *simp*
have *1*:$sum\text{-}id(n,f) \in succ(n) \to succ(m)$ (**is** *?g* ∈ -) **using** *sum-id-tc Forall* **by** *simp*
then have *2*: $arity(p) \leq succ(n)$
using *Forall le-trans*[*of - succ(pred(arity(p)))*] *succpred-leI* **by** *simp*
have $succ(n)\in nat$ $succ(m)\in nat$ **using** *Forall* **by** *auto*
then have *A*:$\bigwedge j .j < succ(n) \Longrightarrow nth(j, Cons(a, \varrho)) = nth(?g`j, Cons(a, \varrho'))$ **if** $a\in M$ **for** *a*
using *that env-coincidence-sum-id Forall ltD* **by** *force*
have *4*:
$sats(M,p,Cons(a,\varrho)) \longleftrightarrow sats(M,ren(p)`succ(n)`succ(m)`?g,Cons(a,\varrho'))$ **if** $a\in M$ **for** *a*
proof −
have *C*:$Cons(a,\varrho) \in list(M)$ $Cons(a,\varrho')\in list(M)$ **using** *Forall that* **by** *auto*
have $sats(M,p,Cons(a,\varrho)) \longleftrightarrow sats(M,ren(p)`succ(n)`succ(m)`?g,Cons(a,\varrho'))$

using *Forall*(*2*)[*OF* ‹$succ(n)\in nat$› ‹$succ(m)\in nat$› *C*(*1*) *C*(*2*) *1 2 A*[*OF* ‹$a\in M$›]] **by** *simp*
then show *?thesis* **.**

```
  qed
  then show ?case using Forall 0 1 2 4 by simp
qed

end
theory Interface
  imports Forcing-Data Relative Internalizations Renaming
begin

lemma Transset-intf :
  Transset(M) ⟹  y∈x ⟹ x ∈ M ⟹ y ∈ M
  by (simp add: Transset-def,auto)

lemma TranssetI :
  (⋀y x. y∈x ⟹ x∈M ⟹ y∈M) ⟹ Transset(M)
  by (auto simp add: Transset-def)

lemma empty-intf :
  infinity-ax(M) ⟹
  (∃ z[M]. empty(M,z))
  by (auto simp add: empty-def infinity-ax-def)

lemma (in forcing-data) zero-in-M:  0 ∈ M
proof −
  from infinity-ax have
        (∃ z[##M]. empty(##M,z))
    by (rule empty-intf)
  then obtain z where
        zm: empty(##M,z)  z∈M
    by auto
  with trans-M have z=0
    by (simp  add: empty-def, blast intro: Transset-intf )
  with zm show ?thesis
      by simp
qed



lemma (in forcing-data) mtriv :
  M-trivial(##M)
  apply (insert trans-M upair-ax Union-ax)
  apply (rule M-trivial.intro)
  apply (simp-all add: zero-in-M)
  apply (rule Transset-intf,simp+)
done

sublocale forcing-data ⊆ M-trivial ##M
  by (rule mtriv)
```

```
abbreviation
 dec10 :: i  (10) where 10 == succ(9)

abbreviation
 dec11 :: i  (11) where 11 == succ(10)

abbreviation
 dec12 :: i  (12) where 12 == succ(11)

abbreviation
 dec13 :: i  (13) where 13 == succ(12)

lemma uniq-dec-2p: <C,D> ∈ M ⟹
            ∀ A∈M. ∀ B∈M. <C,D> = ⟨A, B⟩ ⟶ P(x, A, B)
           ⟷
             P(x, C, D)
  by simp

lemma (in forcing-data) tupling-sep-2p :
   (∀ v∈M. separation(##M,λx. (∀ A∈M. ∀ B∈M. pair(##M,A,B,v) ⟶ Q(x,A,B))))
    ⟷
     (∀ A∈M. ∀ B∈M. separation(##M,λx. Q(x,A,B)))
  apply (simp add: separation-def)
proof (intro ballI iffI)
  fix A B z
  assume
        Eq1: ∀ v∈M. ∀ z∈M. ∃ y∈M. ∀ x∈M. x ∈ y ⟷
              x ∈ z ∧ (∀ A∈M. ∀ B∈M. v = ⟨A, B⟩ ⟶ Q(x, A, B))
     and
        Eq2: A∈M B∈M z∈M
  then have
        Eq3: <A,B>∈M
    by (simp del:setclass-iff add:setclass-iff [symmetric])
  with Eq1 have
             ∀ z∈M. ∃ y∈M. ∀ x∈M. x ∈ y ⟷
              x ∈ z ∧ (∀ C∈M. ∀ D∈M. <A,B> = ⟨C, D⟩ ⟶ Q(x, C, D))
    by (rule bspec)
  with uniq-dec-2p and Eq3 and Eq2 show
             ∃ y∈M. ∀ x∈M. x ∈ y ⟷
              x ∈ z ∧  Q(x, A, B)
    by simp
next
  fix v z
  assume
       asms: v∈M   z∈M
             ∀ A∈M. ∀ B∈M. ∀ z∈M. ∃ y∈M. ∀ x∈M. x ∈ y ⟷ x ∈ z ∧ Q(x, A,
B)
  consider (a) ∃ A∈M. ∃ B∈M. v = ⟨A, B⟩ | (b) ∀ A∈M. ∀ B∈M. v ≠ ⟨A, B⟩ by
```

auto
then show
∃ *y*∈*M*. ∀ *x*∈*M*. *x* ∈ *y* ⟷ *x* ∈ *z* ∧
(∀ *A*∈*M*. ∀ *B*∈*M*. *v* = ⟨*A*, *B*⟩ ⟶ *Q*(*x*, *A*, *B*))
proof *cases*
case *a*
then obtain *A B* **where**
Eq4: *A*∈*M B*∈*M v* = ⟨*A*, *B*⟩
by *auto*
then have
∃ *y*∈*M*. ∀ *x*∈*M*. *x* ∈ *y* ⟷ *x* ∈ *z* ∧ *Q*(*x*, *A*, *B*)
using *asms* **by** *simp*
then show *?thesis* **using** *Eq4* **and** *uniq-dec-2p* **by** *simp*
next
case *b*
then have
∀ *x*∈*M*. *x* ∈ *z* ⟷ *x* ∈ *z* ∧ (∀ *A*∈*M*. ∀ *B*∈*M*. *v* = ⟨*A*, *B*⟩ ⟶ *Q*(*x*, *A*,
B))
by *simp*
then show *?thesis* **using** *b* **and** *asms* **by** *auto*
qed
qed

lemma (**in** *forcing-data*) *tuples-in-M*: *A*∈*M* ⟹ *B*∈*M* ⟹ <*A*,*B*>∈*M*
by (*simp del*:*setclass-iff add*:*setclass-iff* [*symmetric*])

lemma *uniq-dec-5p*: <*A′*,*B′*,*C′*,*D′*,*E′*> ∈ *M* ⟹
∀ *A*∈*M*. ∀ *B*∈*M*. ∀ *C*∈*M*. ∀ *D*∈*M*. ∀ *E*∈*M*. <*A′*,*B′*,*C′*,*D′*,*E′*> =
<*A*,*B*,*C*,*D*,*E*> ⟶
P(*x*,*A*,*B*,*C*,*D*,*E*)
⟷
P(*x*,*A′*,*B′*,*C′*,*D′*,*E′*)
by *simp*

lemma (**in** *forcing-data*) *tupling-sep-5p-aux* :
(∀ *A1*∈*M*. ∀ *A5*∈*M*. ∀ *A4*∈*M*. ∀ *A3*∈*M*. ∀ *A2*∈*M*.
⟨*A4*, *A5*⟩ ∈ *M* ∧ ⟨*A3*, *A4*, *A5*⟩ ∈ *M* ∧ ⟨*A2*, *A3*, *A4*, *A5*⟩ ∈ *M* ∧
v = ⟨*A1*, *A2*, *A3*, *A4*, *A5*⟩ ⟶
Q(*x*, *A1*, *A2*, *A3*, *A4*, *A5*))
⟷
(∀ *A1*∈*M*. ∀ *A2*∈*M*. ∀ *A3*∈*M*. ∀ *A4*∈*M*. ∀ *A5*∈*M*.
v = ⟨*A1*, *A2*, *A3*, *A4*, *A5*⟩ ⟶
Q(*x*, *A1*, *A2*, *A3*, *A4*, *A5*)) **for** *x v*
by (*auto simp add*:*tuples-in-M*)

lemma (**in** *forcing-data*) *tupling-sep-5p* :
$(\forall v{\in}M.\ separation(\#\#M,\lambda x.\ (\forall A1{\in}M.\ \forall A2{\in}M.\ \forall A3{\in}M.\ \forall A4{\in}M.\ \forall A5{\in}M.$

$v = \langle A1, \langle A2, \langle A3, \langle A4, A5\rangle\rangle\rangle\rangle \longrightarrow Q(x,A1,A2,A3,A4,A5))))$
$\longleftrightarrow$
$(\forall A1{\in}M.\ \forall A2{\in}M.\ \forall A3{\in}M.\ \forall A4{\in}M.\ \forall A5{\in}M.\ separation(\#\#M,\lambda x.\ Q(x,A1,A2,A3,A4,A5)))$
proof (*simp add*: *separation-def*, *intro ballI iffI*)
fix *A B C D E z*
assume
Eq1: $\forall v{\in}M.\ \forall z{\in}M.\ \exists y{\in}M.\ \forall x{\in}M.\ x \in y \longleftrightarrow$
$x \in z \wedge (\forall A{\in}M.\ \forall B{\in}M.\ \forall C{\in}M.\ \forall D{\in}M.\ \forall E{\in}M.\ v = \langle A, B,C,D,E\rangle$

$\longrightarrow Q(x, A, B, C, D, E))$
and
Eq2: $A{\in}M$ $B{\in}M$ $C{\in}M$ $D{\in}M$ $E{\in}M$ $z{\in}M$
then have
Eq3: $<A,B,C,D,E>{\in}M$
by (*simp del*:*setclass-iff add*:*setclass-iff* [*symmetric*])
with *Eq1* **have**
$\forall z{\in}M.\ \exists y{\in}M.\ \forall x{\in}M.\ x \in y \longleftrightarrow$
$x \in z \wedge (\forall A'{\in}M.\ \forall B'{\in}M.\ \forall C'{\in}M.\ \forall D'{\in}M.\ \forall E'{\in}M.\ <A,B,C,D,E>$
$= \langle A',B',C',D',E'\rangle$
$\longrightarrow Q(x, A', B', C', D', E'))$
by (*rule bspec*)
with *uniq-dec-5p* **and** *Eq3* **and** *Eq2* **show**
$\exists y{\in}M.\ \forall x{\in}M.\ x \in y \longleftrightarrow$
$x \in z \wedge\ Q(x,A,B,C,D,E)$
by *simp*
next
fix *v z*
assume
asms: $v{\in}M$ $z{\in}M$
$\forall A{\in}M.\ \forall B{\in}M.\ \forall C{\in}M.\ \forall D{\in}M.\ \forall E{\in}M.\ \forall z{\in}M.\ \exists y{\in}M.$
$\forall x{\in}M.\ x \in y \longleftrightarrow x \in z \wedge Q(x, A,B,C,D,E)$
consider (a) $\exists A{\in}M.\ \exists B{\in}M.\ \exists C{\in}M.\ \exists D{\in}M.\ \exists E{\in}M.\ v = \langle A,B,C,D,E\rangle$ |
(b) $\forall A{\in}M.\ \forall B{\in}M.\ \forall C{\in}M.\ \forall D{\in}M.\ \forall E{\in}M.\ v \neq \langle A,B,C,D,E\rangle$ **by** *blast*
then show
$\exists y{\in}M.\ \forall x{\in}M.\ x \in y \longleftrightarrow x \in z \wedge$
$(\forall A{\in}M.\ \forall B{\in}M.\ \forall C{\in}M.\ \forall D{\in}M.\ \forall E{\in}M.\ v = \langle A,B,C,D,E\rangle$
$\longrightarrow Q(x,A,B,C,D,E))$
proof *cases*
case *a*
then obtain *A B C D E* **where**
Eq4: $A{\in}M$ $B{\in}M$ $C{\in}M$ $D{\in}M$ $E{\in}M$ $v = \langle A,B,C,D,E\rangle$
by *auto*
then have
$\exists y{\in}M.\ \forall x{\in}M.\ x \in y \longleftrightarrow x \in z \wedge Q(x,A,B,C,D,E)$
using *asms* **by** *simp*
then show *?thesis* **using** *Eq4* **by** *simp*

next
case *b*
then have
$\forall x{\in}M.\ x \in z \longleftrightarrow x \in z \wedge$
$(\forall A{\in}M.\ \forall B{\in}M.\ \ \forall C{\in}M.\ \forall D{\in}M.\ \forall E{\in}M.\ v = \langle A,B,C,D,E\rangle \longrightarrow$
$Q(x,A,B,C,D,E))$
by *simp*
then show *?thesis* **using** *b* **and** *asms* **by** *auto*
qed
qed

lemma (**in** *forcing-data*) *tupling-sep-5p-rel* :
$(\forall v{\in}M.\ separation(\#\#M,\lambda x.\ (\forall A1{\in}M.\ \forall A5{\in}M.\ \forall A4{\in}M.\ \forall A3{\in}M.\ \forall A2{\in}M.$

$\forall B1{\in}M.\ \forall B2{\in}M.\ \forall B3{\in}M.$
$pair(\#\#M,A4,A5,B1)$ &
$pair(\#\#M,A3,B1,B2)$ &
$pair(\#\#M,A2,B2,B3)$ &
$pair(\#\#M,A1,B3,v)$
$\longrightarrow Q(x,A1,A2,A3,A4,A5))))$
$\longleftrightarrow$
$(\forall A1{\in}M.\ \forall A5{\in}M.\ \forall A4{\in}M.\ \forall A3{\in}M.\ \forall A2{\in}M.\ separation(\#\#M,\lambda x.\ Q(x,A1,A2,A3,A4,A5)))$
proof (*simp*)
have
$(\forall A1{\in}M.\ \forall A5{\in}M.\ \forall A4{\in}M.\ \forall A3{\in}M.\ \forall A2{\in}M.$
$\langle A4,\ A5\rangle \in M \wedge \langle A3,\ A4,\ A5\rangle \in M \wedge \langle A2,\ A3,\ A4,\ A5\rangle \in M \wedge v =$
$\langle A1,\ A2,\ A3,\ A4,\ A5\rangle \longrightarrow$
$Q(x,\ A1,\ A2,\ A3,\ A4,\ A5))$
$\longleftrightarrow$
$(\forall A1{\in}M.\ \forall A2{\in}M.\ \forall A3{\in}M.\ \forall A4{\in}M.\ \forall A5{\in}M.$
$v = \langle A1,\ A2,\ A3,\ A4,\ A5\rangle \longrightarrow$
$Q(x,\ A1,\ A2,\ A3,\ A4,\ A5))$ **for** *x v*
by (*rule tupling-sep-5p-aux*)
then have
$(\forall v{\in}M.\ separation$
$(\#\#M,$
$\lambda x.\ \forall A1{\in}M.\ \forall A5{\in}M.\ \forall A4{\in}M.\ \forall A3{\in}M.\ \forall A2{\in}M.$
$\langle A4,\ A5\rangle \in M \wedge \langle A3,\ A4,\ A5\rangle \in M \wedge \langle A2,\ A3,\ A4,\ A5\rangle \in M \wedge v =$
$\langle A1,\ A2,\ A3,\ A4,\ A5\rangle \longrightarrow$
$Q(x,\ A1,\ A2,\ A3,\ A4,\ A5)))$
$\longleftrightarrow$
$(\forall v{\in}M.\ separation$
$(\#\#M,$
$\lambda x.\ \forall A1{\in}M.\ \forall A2{\in}M.\ \forall A3{\in}M.\ \forall A4{\in}M.\ \forall A5{\in}M.$
$v = \langle A1,\ A2,\ A3,\ A4,\ A5\rangle \longrightarrow$
$Q(x,\ A1,\ A2,\ A3,\ A4,\ A5)))$
by *simp*
also have

... $\longleftrightarrow$
($\forall$ *A1*$\in$*M*. $\forall$ *A2*$\in$*M*. $\forall$ *A3*$\in$*M*. $\forall$ *A4*$\in$*M*. $\forall$ *A5*$\in$*M*. *separation*(##*M*,λ*x*. *Q*(*x*,*A1*,*A2*,*A3*,*A4*,*A5*)))
using *tupling-sep-5p* **by** *simp*
finally show
($\forall$ *v*$\in$*M*. *separation*
(##*M*,
λ*x*. $\forall$ *A1*$\in$*M*. $\forall$ *A5*$\in$*M*. $\forall$ *A4*$\in$*M*. $\forall$ *A3*$\in$*M*. $\forall$ *A2*$\in$*M*.
$\langle$*A4*, *A5*$\rangle$ $\in$ *M* $\wedge$ $\langle$*A3*, *A4*, *A5*$\rangle$ $\in$ *M* $\wedge$ $\langle$*A2*, *A3*, *A4*, *A5*$\rangle$ $\in$ *M* $\wedge$ *v* = $\langle$*A1*, *A2*,
A3, *A4*, *A5*$\rangle$ $\longrightarrow$
Q(*x*, *A1*, *A2*, *A3*, *A4*, *A5*))) $\longleftrightarrow$
($\forall$ *A1*$\in$*M*. $\forall$ *A5*$\in$*M*. $\forall$ *A4*$\in$*M*. $\forall$ *A3*$\in$*M*. $\forall$ *A2*$\in$*M*. *separation*(##*M*, λ*x*. *Q*(*x*,
A1, *A2*, *A3*, *A4*, *A5*)))
by *auto*
qed

lemma (**in** *forcing-data*) *tupling-sep-5p-rel2* :
($\forall$ *v*$\in$*M*. *separation*(##*M*,λ*x*. ($\forall$ *B3*$\in$*M*. $\forall$ *B2*$\in$*M*. $\forall$ *B1*$\in$*M*.
$\forall$ *A5*$\in$*M*. $\forall$ *A4*$\in$*M*. $\forall$ *A3*$\in$*M*. $\forall$ *A2*$\in$*M*. $\forall$ *A1*$\in$*M*.
pair(##*M*,*A4*,*A5*,*B1*) &
pair(##*M*,*A3*,*B1*,*B2*) &
pair(##*M*,*A2*,*B2*,*B3*) &
pair(##*M*,*A1*,*B3*,*v*)
$\longrightarrow$ *Q*(*x*,*A1*,*A2*,*A3*,*A4*,*A5*))))
$\longleftrightarrow$
($\forall$ *A5*$\in$*M*. $\forall$ *A4*$\in$*M*. $\forall$ *A3*$\in$*M*. $\forall$ *A2*$\in$*M*. $\forall$ *A1*$\in$*M*. *separation*(##*M*,λ*x*. *Q*(*x*,*A1*,*A2*,*A3*,*A4*,*A5*)))
proof –
have
($\forall$ *B3*$\in$*M*. $\forall$ *B2*$\in$*M*. $\forall$ *B1*$\in$*M*.
$\forall$ *A5*$\in$*M*. $\forall$ *A4*$\in$*M*. $\forall$ *A3*$\in$*M*. $\forall$ *A2*$\in$*M*. $\forall$ *A1*$\in$*M*.
pair(##*M*,*A4*,*A5*,*B1*) &
pair(##*M*,*A3*,*B1*,*B2*) &
pair(##*M*,*A2*,*B2*,*B3*) &
pair(##*M*,*A1*,*B3*,*v*)
$\longrightarrow$ *Q*(*x*,*A1*,*A2*,*A3*,*A4*,*A5*))
$\longleftrightarrow$
($\forall$ *A1*$\in$*M*. $\forall$ *A5*$\in$*M*. $\forall$ *A4*$\in$*M*. $\forall$ *A3*$\in$*M*. $\forall$ *A2*$\in$*M*.
$\forall$ *B1*$\in$*M*. $\forall$ *B2*$\in$*M*. $\forall$ *B3*$\in$*M*.
pair(##*M*,*A4*,*A5*,*B1*) &
pair(##*M*,*A3*,*B1*,*B2*) &
pair(##*M*,*A2*,*B2*,*B3*) &
pair(##*M*,*A1*,*B3*,*v*)
$\longrightarrow$ *Q*(*x*,*A1*,*A2*,*A3*,*A4*,*A5*))
(**is** *?P*$\longleftrightarrow$*?Q*) **for** *x v*
by *auto*
then have
separation(##*M*,λ*x*. *?P*(*x*,*v*)) $\longleftrightarrow$ *separation*(##*M*,λ*x*. *?Q*(*x*,*v*)) **for** *v*
by *auto*
then have
($\forall$ *v*$\in$*M*. *separation*(##*M*,λ*x*. *?P*(*x*,*v*)))

$\longleftrightarrow$
(∀ *v*∈*M*. *separation*(##*M*,λ*x*. ?*Q*(*x*,*v*)))
by *blast*
also have
... ⟷ (∀ *A1*∈*M*. ∀ *A5*∈*M*. ∀ *A4*∈*M*. ∀ *A3*∈*M*. ∀ *A2*∈*M*. *separation*(##*M*,λ*x*. *Q*(*x*,*A1*,*A2*,*A3*,*A4*,*A5*)))
using *tupling-sep-5p-rel* **by** *simp*
finally show *?thesis* **by** *auto*
qed

definition
tupling-fm-2p :: *i* ⇒ *i* **where**
tupling-fm-2p(φ) = *Forall*(*Forall*(*Implies*(*pair-fm*(*1*,*0*,*3*),φ)))

lemma [*TC*] : ⟦ φ ∈ *formula* ⟧ ⟹ *tupling-fm-2p*(φ) ∈ *formula*
by (*simp add*: *tupling-fm-2p-def*)

lemma *arity-tup2p* :
⟦ φ ∈ *formula* ; *arity*(φ) = *3* ⟧ ⟹ *arity*(*tupling-fm-2p*(φ)) = *2*
by (*simp add*: *tupling-fm-2p-def arity-incr-bv-lemma pair-fm-def upair-fm-def Un-commute nat-union-abs1*)

definition
tupling-fm-5p :: *i* ⇒ *i* **where**
tupling-fm-5p(φ) =
Forall(*Forall*(*Forall*(*Forall*(*Forall*(*Forall*(*Forall*(*Forall*(
Implies(*And*(*pair-fm*(*3*,*4*,*5*),
And(*pair-fm*(*2*,*5*,*6*),
And(*pair-fm*(*1*,*6*,*7*),
pair-fm(*0*,*7*,*9*)))),φ)))))))))

lemma [*TC*] : ⟦ φ ∈ *formula* ⟧ ⟹ *tupling-fm-5p*(φ) ∈ *formula*
by (*simp add*: *tupling-fm-5p-def*)

lemma *arity-tup5p* :
⟦ φ ∈ *formula* ; *arity*(φ) = *9* ⟧ ⟹ *arity*(*tupling-fm-5p*(φ)) = *2*
by (*simp add*: *tupling-fm-5p-def arity-incr-bv-lemma pair-fm-def upair-fm-def Un-commute nat-union-abs1*)

lemma *leq-9*:
n≤*9* ⟹ *n*=*0* | *n*=*1* | *n*=*2* | *n*=*3* | *n*=*4* | *n*=*5* | *n*=*6*| *n*=*7* | *n*=*8* | *n*=*9*
by (*clarsimp simp add*:*not-lt-iff-le*, *auto simp add*:*lt-def*)

lemma *arity-tup5p-leq* :
⟦ φ ∈ *formula* ; *arity*(φ) ≤ *9* ⟧ ⟹ *arity*(*tupling-fm-5p*(φ)) = *2*
by (*drule leq-9*, *elim disjE*, *simp-all add*:*tupling-fm-5p-def arity-incr-bv-lemma pair-fm-def*

upair-fm-def Un-commute nat-union-abs1)

lemma *arity-inter-fm* :
arity(*Forall*(*Implies*(*Member*(*0*,*2*),*Member*(*1*,*0*)))) = *2*
by (*simp add*: *Un-commute nat-union-abs1*)

lemma (**in** *forcing-data*) *inter-sep-intf* :
assumes
A∈*M*
shows
separation(##*M*,λ*x* . ∀ *y*∈*M* . *y*∈*A* ⟶ *x*∈*y*)
proof −
from *separation-ax arity-inter-fm* **have**
∀ *a*∈*M*. *separation*(##*M*, λ*x*. *sats*(*M*, *Forall*(*Implies*(*Member*(*0*,*2*),*Member*(*1*,*0*))),
[*x*, *a*]))
by *simp*
with ‹*A*∈*M*› **have**
separation(##*M*, λ*x*. *sats*(*M*, *Forall*(*Implies*(*Member*(*0*,*2*),*Member*(*1*,*0*))),
[*x*, *A*]))
by *simp*
with ‹*A*∈*M*› **show** *?thesis* **unfolding** *separation-def* **by** *simp*
qed

lemma *arity-diff-fm*:
arity(*Neg*(*Member*(*0*,*1*))) = *2*
by (*simp add*: *nat-union-abs1*)

lemma (**in** *forcing-data*) *diff-sep-intf* :
assumes
B∈*M*
shows
separation(##*M*,λ*x* . *x*∉*B*)
proof −
from *separation-ax arity-diff-fm* **have**
∀ *a*∈*M*. *separation*(##*M*, λ*x*. *sats*(*M*, *Neg*(*Member*(*0*,*1*)), [*x*, *a*]))
by *simp*
with ‹*B*∈*M*› **have**
separation(##*M*, λ*x*. *sats*(*M*, *Neg*(*Member*(*0*,*1*)), [*x*, *B*]))
by *simp*
with ‹*B*∈*M*› **show** *?thesis* **unfolding** *separation-def* **by** *simp*
qed

definition
cartprod-sep-fm :: *i* **where**
cartprod-sep-fm ==
Exists(*And*(*Member*(*0*,*2*),
Exists(*And*(*Member*(*0*,*2*),*pair-fm*(*1*,*0*,*4*)))))

lemma *cartprof-sep-fm-type* [*TC*] :
cartprod-sep-fm ∈ *formula*
by (*simp add*: *cartprod-sep-fm-def*)

lemma *arity-cartprod-fm* [*simp*] : *arity*(*cartprod-sep-fm*) = *3*
by (*simp add*: *cartprod-sep-fm-def pair-fm-def upair-fm-def*
Un-commute nat-union-abs1)

lemma (**in** *forcing-data*) *cartprod-sep-intf* :
assumes
A∈*M*
and
B∈*M*
shows
separation(##*M*,λ*z*. ∃ *x*∈*M*. *x*∈*A* ∧ (∃ *y*∈*M*. *y*∈*B* ∧ *pair*(##*M*,*x*,*y*,*z*)))
proof −
from *separation-ax arity-tup2p* **have**
(∀ *v*∈*M*. *separation*(##*M*,λ*x*. *sats*(*M*,*tupling-fm-2p*(*cartprod-sep-fm*),[*x*,*v*])))
by *simp*
then have
(∀ *v*∈*M*. *separation*(##*M*, λ*x*. ∀ *A*∈*M*. ∀ *B*∈*M*. *pair*(##*M*, *A*, *B*, *v*) ⟶
(∃ *xa*∈*M*. *xa* ∈ *A* ∧ (∃ *y*∈*M*. *y* ∈ *B* ∧ *pair*(##*M*, *xa*, *y*, *x*)))))
unfolding *separation-def tupling-fm-2p-def cartprod-sep-fm-def* **by** (*simp del*: *pair-abs*)
with *tupling-sep-2p* **have**
(∀ *A*∈*M*. ∀ *B*∈*M*. *separation*(##*M*, λ*z*. ∃ *x*∈*M*. *x* ∈ *A* ∧ (∃ *y*∈*M*. *y* ∈ *B* ∧ *pair*(##*M*, *x*, *y*, *z*))))
by *simp*
with ‹*A*∈*M*› ‹*B*∈*M*› **show** *?thesis* **by** *simp*
qed

definition
image-sep-fm :: *i* **where**
image-sep-fm ==
Exists(*And*(*Member*(*0*,*1*),
Exists(*And*(*Member*(*0*,*3*),*pair-fm*(*0*,*4*,*1*)))))

lemma *image-sep-fm-type* [*TC*] :
image-sep-fm ∈ *formula*

by (*simp add*: *image-sep-fm-def*)

lemma [*simp*] : *arity*(*image-sep-fm*) = *3*
by (*simp add*: *image-sep-fm-def pair-fm-def upair-fm-def*
Un-commute nat-union-abs1)

lemma (**in** *forcing-data*) *image-sep-intf* :
assumes
A∈*M*
and
r∈*M*
shows
separation(##*M*, λ*y*. ∃ *p*∈*M*. *p*∈*r* & (∃ *x*∈*M*. *x*∈*A* & *pair*(##*M*,*x*,*y*,*p*)))
proof −
from *separation-ax arity-tup2p* **have**
(∀ *v*∈*M*. *separation*(##*M*,λ*x*. *sats*(*M*,*tupling-fm-2p*(*image-sep-fm*),[*x*,*v*])))
by *simp*
then have
(∀ *v*∈*M*. *separation*(##*M*, λ*x*. ∀ *A*∈*M*. ∀ *B*∈*M*. *pair*(##*M*, *A*, *B*, *v*) ⟶
(∃ *p*∈*M*. *p* ∈ *B* ∧ (∃ *xa*∈*M*. *xa* ∈ *A* ∧ *pair*(##*M*, *xa*, *x*, *p*)))))
unfolding *separation-def tupling-fm-2p-def image-sep-fm-def* **by** (*simp del*: *pair-abs*)
with *tupling-sep-2p* **have**
(∀ *A*∈*M*. ∀ *r*∈*M*. *separation*(##*M*, λ*y*. ∃ *p*∈*M*. *p*∈*r* & (∃ *x*∈*M*. *x*∈*A* & *pair*(##*M*,*x*,*y*,*p*))))
by *simp*
with ‹*A*∈*M*› ‹*r*∈*M*› **show** *?thesis* **by** *simp*
qed

definition
converse-sep-fm :: *i* **where**
converse-sep-fm ==
Exists(*And*(*Member*(*0*,*2*),
Exists(*Exists*(*And*(*pair-fm*(*1*,*0*,*2*),*pair-fm*(*0*,*1*,*3*))))))

lemma *converse-sep-fm-type* [*TC*] : *converse-sep-fm* ∈ *formula*
by (*simp add*: *converse-sep-fm-def*)

lemma [*simp*] : *arity*(*converse-sep-fm*) = *2*
by (*simp add*: *converse-sep-fm-def pair-fm-def upair-fm-def*
Un-commute nat-union-abs1)

lemma (**in** *forcing-data*) *converse-sep-intf* :
assumes
R∈*M*
shows
separation(##*M*,λ*z*. ∃ *p*∈*M*. *p*∈*R* & (∃ *x*∈*M*.∃ *y*∈*M*. *pair*(##*M*,*x*,*y*,*p*) &
pair(##*M*,*y*,*x*,*z*)))
proof −

from *separation-ax* **have**
∀ *r*∈*M*. *separation*(##*M*, λ*x*. *sats*(*M*,*converse-sep-fm*, [*x*, *r*]))
by *simp*
with ‹*R*∈*M*› **have**
separation(##*M*, λ*x*. *sats*(*M*,*converse-sep-fm*, [*x*, *R*]))
by *simp*
with ‹*R*∈*M*› **show** *?thesis* **unfolding** *separation-def converse-sep-fm-def* **by** (*simp del*: *pair-abs*)
qed

definition
restrict-sep-fm :: *i* **where**
restrict-sep-fm == *Exists*(*And*(*Member*(*0*,*2*),*Exists*(*pair-fm*(*1*,*0*,*2*))))

lemma *restrict-sep-fm-type* [*TC*] : *restrict-sep-fm* ∈ *formula*
by (*simp add*: *restrict-sep-fm-def*)

lemma [*simp*] : *arity*(*restrict-sep-fm*) = *2*
by (*simp add*: *restrict-sep-fm-def pair-fm-def upair-fm-def*
Un-commute nat-union-abs1)

lemma (**in** *forcing-data*) *restrict-sep-intf* :
assumes
A∈*M*
shows
separation(##*M*,λ*z*. ∃ *x*∈*M*. *x*∈*A* & (∃ *y*∈*M*. *pair*(##*M*,*x*,*y*,*z*)))
proof −
from *separation-ax* **have**
∀ *a*∈*M*. *separation*(##*M*, λ*x*. *sats*(*M*,*restrict-sep-fm*, [*x*, *a*]))
by *simp*
with ‹*A*∈*M*› **have**
separation(##*M*, λ*x*. *sats*(*M*,*restrict-sep-fm*, [*x*, *A*]))
by *simp*
with ‹*A*∈*M*› **show** *?thesis* **unfolding** *separation-def restrict-sep-fm-def* **by** (*simp del*: *pair-abs*)
qed

definition
comp-sep-fm :: *i* **where**
comp-sep-fm ==
Exists(*Exists*(*Exists*(*Exists*(*Exists*
(*And*(*pair-fm*(*4*,*2*,*7*),*And*(*pair-fm*(*4*,*3*,*1*),
And(*pair-fm*(*3*,*2*,*0*),*And*(*Member*(*1*,*5*),*Member*(*0*,*6*))))))))))

lemma *comp-sep-fm-type* [*TC*] : *comp-sep-fm* ∈ *formula*
by (*simp add*: *comp-sep-fm-def*)

lemma [*simp*] : *arity*(*comp-sep-fm*) = *3*
by (*simp add*: *comp-sep-fm-def pair-fm-def upair-fm-def Un-commute nat-union-abs1*)

lemma (**in** *forcing-data*) *comp-sep-intf* :
assumes
R∈*M*
and
S∈*M*
shows
separation(##*M*,λ*xz*. ∃ *x*∈*M*. ∃ *y*∈*M*. ∃ *z*∈*M*. ∃ *xy*∈*M*. ∃ *yz*∈*M*.
pair(##*M*,*x*,*z*,*xz*) & *pair*(##*M*,*x*,*y*,*xy*) & *pair*(##*M*,*y*,*z*,*yz*) & *xy*∈*S*
& *yz*∈*R*)
proof −
from *separation-ax arity-tup2p* **have**
(∀ *v*∈*M*. *separation*(##*M*,λ*x*. *sats*(*M*,*tupling-fm-2p*(*comp-sep-fm*),[*x*,*v*])))
by *simp*
then have
(∀ *v*∈*M*. *separation*
(##*M*, λ*x*. ∀ *A*∈*M*. ∀ *B*∈*M*. *pair*(##*M*, *A*, *B*, *v*) ⟶
(∃ *xa*∈*M*. ∃ *y*∈*M*. ∃ *z*∈*M*. ∃ *xy*∈*M*. ∃ *yz*∈*M*. *pair*(##*M*,
xa, *z*, *x*) ∧
pair(##*M*, *xa*, *y*, *xy*) ∧ *pair*(##*M*, *y*, *z*, *yz*) ∧ *xy* ∈ *B* ∧ *yz* ∈ *A*)))
unfolding *separation-def tupling-fm-2p-def comp-sep-fm-def* **by** (*simp del*: *pair-abs*)
with *tupling-sep-2p* **have**
(∀ *r*∈*M*. ∀ *s*∈*M*. *separation*
(##*M*, λ*xz*. ∃ *x*∈*M*. ∃ *y*∈*M*. ∃ *z*∈*M*. ∃ *xy*∈*M*. ∃ *yz*∈*M*. *pair*(##*M*, *x*, *z*,
xz) ∧
pair(##*M*, *x*, *y*, *xy*) ∧ *pair*(##*M*, *y*, *z*, *yz*) ∧ *xy* ∈
s ∧ *yz* ∈ *r*))
by *simp*
with ‹*S*∈*M*› ‹*R*∈*M*› **show** *?thesis* **by** *simp*
qed

lemma *arity-pred-fm* [*simp*] :
arity(*Exists*(*And*(*Member*(*0*,*2*),*pair-fm*(*3*,*1*,*0*)))) = *3*
by (*simp add*: *pair-fm-def upair-fm-def Un-commute nat-union-abs1*)

lemma (**in** *forcing-data*) *pred-sep-intf*:
assumes
R∈*M*
and
X∈*M*
shows
separation(##*M*, λ*y*. ∃ *p*∈*M*. *p*∈*R* & *pair*(##*M*,*y*,*X*,*p*))
proof −
from *separation-ax arity-tup2p arity-pred-fm* **have**
(∀ *v*∈*M*. *separation*(##*M*,λ*x*. *sats*(*M*,*tupling-fm-2p*(*Exists*(*And*(*Member*(*0*,*2*),

pair-fm(*3*,*1*,*0*)))),[*x*,*v*])))
by *simp*
then have
(∀ *v*∈*M*. *separation*(##*M*, λ*x*. ∀ *A*∈*M*. ∀ *B*∈*M*. *pair*(##*M*, *A*, *B*, *v*) ⟶
(∃ *p*∈*M*. *p* ∈ *A* ∧ *pair*(##*M*, *x*, *B*,
p))))
unfolding *separation-def tupling-fm-2p-def* **by** (*simp del*: *pair-abs*)
with *tupling-sep-2p* **have**
∀ *r*∈*M*. ∀ *x*∈*M*. *separation*(##*M*, λ*y*. ∃ *p*∈*M*. *p*∈*r* & *pair*(##*M*,*y*,*x*,*p*))
by *simp*
with ‹*R*∈*M*› ‹*X*∈*M*› **show** *?thesis* **by** *simp*
qed

definition
memrel-fm :: *i* **where**
memrel-fm == *Exists*(*Exists*(*And*(*pair-fm*(*1*,*0*,*2*),*Member*(*1*,*0*))))

lemma [*TC*] : *memrel-fm* ∈ *formula*
by (*simp add*: *memrel-fm-def*)

lemma [*simp*] : *arity*(*memrel-fm*) = *1*
by (*simp add*: *memrel-fm-def pair-fm-def upair-fm-def Un-commute nat-union-abs1*)

lemma (**in** *forcing-data*) *memrel-sep-intf*:
separation(##*M*, λ*z*. ∃ *x*∈*M*. ∃ *y*∈*M*. *pair*(##*M*,*x*,*y*,*z*) & *x* ∈ *y*)
proof −
from *separation-ax* **have**
(∀ *v*∈*M*. *separation*(##*M*,λ*x*. *sats*(*M*,*memrel-fm*,[*x*,*v*])))
by *simp*
then have
(∀ *v*∈*M*. *separation*(##*M*, λ*z*. ∃ *x*∈*M*. ∃ *y*∈*M*. *pair*(##*M*,*x*,*y*,*z*) & *x* ∈ *y*))
unfolding *separation-def memrel-fm-def* **by** (*simp del*: *pair-abs*)
with *zero-in-M* **show** *?thesis* **by** *auto*
qed

definition
is-recfun-sep-fm :: *i* **where**
is-recfun-sep-fm ==
Exists(*Exists*(*And*(*pair-fm*(*10*,*3*,*1*),*And*(*Member*(*1*,*6*),*And*(*pair-fm*(*10*,*2*,*0*),*And*(*Member*(*0*,*6*),
Exists(*Exists*(*And*(*fun-apply-fm*(*7*,*12*,*1*),
And(*fun-apply-fm*(*6*,*12*,*0*),*Neg*(*Equal*(*1*,*0*))))))))))))

lemma *is-recfun-sep-fm* [*TC*] : *is-recfun-sep-fm* ∈ *formula*
by (*simp add*: *is-recfun-sep-fm-def*)

lemma [*simp*] : *arity*(*is-recfun-sep-fm*) = *9*
  by (*simp add*: *is-recfun-sep-fm-def fun-apply-fm-def upair-fm-def*
      image-fm-def big-union-fm-def pair-fm-def Un-commute nat-union-abs1)

lemma (**in** *forcing-data*) *is-recfun-sep-intf* :
  assumes
    r∈*M f*∈*M g*∈*M a*∈*M b*∈*M*
  shows
    separation(##*M*,λ*x*. ∃ *xa*∈*M*. ∃ *xb*∈*M*.
        pair(##*M*,*x*,*a*,*xa*) & *xa* ∈ *r* & *pair*(##*M*,*x*,*b*,*xb*) & *xb* ∈ *r* &
      (∃ *fx*∈*M*. ∃ *gx*∈*M*. *fun-apply*(##*M*,*f*,*x*,*fx*) & *fun-apply*(##*M*,*g*,*x*,*gx*)
&
          fx ≠ *gx*))
proof −
  from *separation-ax arity-tup5p* **have**
    (∀ *v*∈*M*. *separation*(##*M*,λ*x*. *sats*(*M*,*tupling-fm-5p*(*is-recfun-sep-fm*),[*x*,*v*])))
    by *simp*
  then have
    (∀ *v*∈*M*. *separation*
      (##*M*, λ*x*. ∀ *B3*∈*M*. ∀ *B2*∈*M*. ∀ *B1*∈*M*. ∀ *A5*∈*M*. ∀ *A4*∈*M*. ∀ *A3*∈*M*.
∀ *A2*∈*M*.
        ∀ *A1*∈*M*. *pair*(##*M*, *A4*, *A5*, *B1*) ∧ *pair*(##*M*, *A3*, *B1*, *B2*)
∧ *pair*(##*M*, *A2*, *B2*, *B3*) ∧
          pair(##*M*, *A1*, *B3*, *v*) ⟶
      (∃ *xa*∈*M*. ∃ *xb*∈*M*. *pair*(##*M*, *x*, *A2*, *xa*) ∧ *xa* ∈ *A5* ∧ *pair*(##*M*, *x*,
A1, *xb*) ∧ *xb* ∈ *A5* ∧
        (∃ *fx*∈*M*. ∃ *gx*∈*M*. *fun-apply*(##*M*, *A4*, *x*, *fx*) ∧ *fun-apply*(##*M*, *A3*,
x, *gx*) ∧ *fx* ≠ *gx*))))
    unfolding *separation-def tupling-fm-5p-def is-recfun-sep-fm-def* **by** (*simp del*:
pair-abs)
  with *tupling-sep-5p-rel2* **have**
    (∀ *r*∈*M*. ∀ *f*∈*M*. ∀ *g*∈*M*. ∀ *a*∈*M*. ∀ *b*∈*M*.
    separation(##*M*,λ*x*. ∃ *xa*∈*M*. ∃ *xb*∈*M*.
        pair(##*M*,*x*,*a*,*xa*) & *xa* ∈ *r* & *pair*(##*M*,*x*,*b*,*xb*) & *xb* ∈ *r* &
      (∃ *fx*∈*M*. ∃ *gx*∈*M*. *fun-apply*(##*M*,*f*,*x*,*fx*) & *fun-apply*(##*M*,*g*,*x*,*gx*)
&
          fx ≠ *gx*)))
  by *simp*
  with ‹*r*∈*M*› ‹*f*∈*M*› ‹*g*∈*M*› ‹*a*∈*M*› ‹*b*∈*M*› **show** *?thesis* **by** *simp*
qed

definition
  sixp-sep-perm :: *i* **where**
  sixp-sep-perm == {<*0*,*8*>,<*1*,*0*>,<*2*,*1*>,<*3*,*2*>,<*4*,*3*>,<*5*,*4*>}

lemma *sixp-perm-ftc* : *sixp-sep-perm* ∈ *6* −||> *9*
  by(*unfold sixp-sep-perm-def*,(*rule consI*,*auto*)+,*rule emptyI*)

lemma *dom-sixp-perm* : *domain*(*sixp-sep-perm*) = *6*
  by(*unfold sixp-sep-perm-def* ,*auto*)

lemma *sixp-perm-tc* : *sixp-sep-perm* ∈ *6* → *9*
  by(*subst dom-sixp-perm*[*symmetric*],*rule FiniteFun-is-fun*,*rule sixp-perm-ftc*)

lemma *apply-fun*: *f* ∈ *Pi*(*A*,*B*) ==> <*a*,*b*>: *f* ⟹ *f*'*a* = *b*
  by(*auto simp add*: *apply-iff*)

lemma *sixp-perm-env* :
  {*x*,*a1* ,*a2* ,*a3* ,*a4* ,*a5*} ⊆ *A* ⟹ *j*<*6* ⟹
  nth(*j*,[*x*,*a1* ,*a2* ,*a3* ,*a4* ,*a5*]) = *nth*(*sixp-sep-perm*'*j*,[*a1* ,*a2* ,*a3* ,*a4* ,*a5* ,*b1* ,*b2* ,*b3* ,*x*,*v*])
  apply(*subgoal-tac j*∈*nat*)
  apply(*rule natE*,*simp*,*subst apply-fun*,*rule sixp-perm-tc*,*simp add*:*sixp-sep-perm-def* ,*simp*+)+
  apply(*subst apply-fun*,*rule sixp-perm-tc*,*simp add*:*sixp-sep-perm-def* ,*simp*+,*drule ltD*,*auto*)
  done

lemma (**in** *forcing-data*) *sixp-sep*:
  assumes
    φ ∈ *formula arity*(φ)≤*6 a1*∈*M a2*∈*M a3*∈*M a4*∈*M a5*∈*M*
  shows
    separation(##*M*,λ*x*. *sats*(*M*,φ,[*x*,*a1* ,*a2* ,*a3* ,*a4* ,*a5*]))
proof −
  let
    ?f =*sixp-sep-perm*
  let
    ?φ′=*ren*(φ)'*6*'*9*'*?f*
  from *assms* **have**
    arity(*?φ*′)≤*9 ?φ*′ ∈ *formula*
    using *sixp-perm-tc ren-arity ren-tc* **by** *simp-all*
  then have
    (∀ *v*∈*M*. *separation*(##*M*,λ*x*. *sats*(*M*,*tupling-fm-5p*(*?φ*′),[*x*,*v*])))
    using *separation-ax arity-tup5p-leq* **by** *simp*
  then have
    Eq1: (∀ *v*∈*M*. *separation*
        (##*M*, λ*x*. ∀ *B3*∈*M*. ∀ *B2*∈*M*. ∀ *B1*∈*M*. ∀ *A5*∈*M*. ∀ *A4*∈*M*. ∀ *A3*∈*M*. ∀ *A2*∈*M*.
            ∀ *A1*∈*M*. *pair*(##*M*, *A4*, *A5*, *B1*) ∧ *pair*(##*M*, *A3*, *B1*, *B2*) ∧ *pair*(##*M*, *A2*, *B2*, *B3*) ∧
                pair(##*M*, *A1*, *B3*, *v*) ⟶
          sats(*M*,*?φ*′,[*A1* ,*A2* ,*A3* ,*A4* ,*A5* ,*B1* ,*B2* ,*B3* ,*x*,*v*])))
    (**is** ∀ *v*∈*M*. *separation*(- , λ*x*. *?P*(*x*,*v*)))
    unfolding *separation-def tupling-fm-5p-def* **by** (*simp del*: *pair-abs*)
  {
    fix *B1 B2 B3 A1 A2 A3 A4 A5 x v*
    assume
      x∈*M v*∈*M*

B3∈M B2∈M B1∈M A5∈M A4∈M A3∈M A2∈M A1∈M
with *assms* **have**
sats(M,?φ′,[A1,A2,A3,A4,A5,B1,B2,B3,x,v]) ⟷ *sats(M,φ,[x,A1,A2,A3,A4,A5])*

(**is** *sats(-,-,?env1)* ⟷ *sats(-,-,?env2)*)
using *sats-iff-sats-ren[of φ 6 9 ?env2 M ?env1 ?f] sixp-perm-tc sixp-perm-env*
[of - - - - - - M]
by *auto*
}
then have
Eq2: *x∈M* ⟹ *v∈M* ⟹ *?P(x,v)* ⟷ (∀ *B3∈M*. ∀ *B2∈M*. ∀ *B1∈M*. ∀ *A5∈M*. ∀ *A4∈M*. ∀ *A3∈M*. ∀ *A2∈M*.
∀ *A1∈M*. *pair(##M, A4, A5, B1)* ∧ *pair(##M, A3, B1, B2)* ∧ *pair(##M, A2, B2, B3)* ∧
pair(##M, A1, B3, v) ⟶
sats(M,φ,[x,A1,A2,A3,A4,A5])) (**is** - ⟹ -⟹ - ⟷ *?Q(x,v)*) **for** *x v*
by (*simp del*: *pair-abs*)
define *PP* **where** *PP* ≡ *?P*
define *QQ* **where** *QQ* ≡ *?Q*
from *Eq2* **have**
x∈M ⟹ *v∈M* ⟹ *PP(x,v)* ⟷ *QQ(x,v)* **for** *x v*
unfolding *PP-def QQ-def* **.**
then have
v∈M ⟹
(∀ *z[##M]*. ∃ *y[##M]*. ∀ *x[##M]*. *x* ∈ *y* ⟷ *x* ∈ *z* ∧ *PP(x,v)*) ⟷
(∀ *z[##M]*. ∃ *y[##M]*. ∀ *x[##M]*. *x* ∈ *y* ⟷ *x* ∈ *z* ∧ *QQ(x,v)*) **for** *v* **by** (*simp del*: *pair-abs*)
with *Eq1* **have**
(∀ *v∈M*. *separation* (*##M*, λ*x*. *QQ(x,v)*))
unfolding *separation-def PP-def* **by** (*simp del*: *pair-abs*)
with *assms* **show** *?thesis* **unfolding** *QQ-def* **using** *tupling-sep-5p-rel2* **by** *simp*
qed

definition
is-cons-fm :: *i* ⇒ *i* ⇒ *i* ⇒ *i* **where**
is-cons-fm(a,b,z) == Exists(And(upair-fm(succ(a),succ(a),0),union-fm(0,succ(b),succ(z))))

lemma *is-cons-type* [*TC*]:
[| *x* ∈ *nat*; *y* ∈ *nat*; *z* ∈ *nat* |] ==> *is-cons-fm(x,y,z)* ∈ *formula*
by (*simp add*: *is-cons-fm-def*)

lemma *is-cons-fm* [*simp*] :
⟦ *a* ∈ *nat* ; *b* ∈ *nat* ; *z* ∈ *nat* ; *env* ∈ *list(A)* ⟧ ⟹
sats(A,is-cons-fm(a,b,z),env) ⟷
is-cons(##A,nth(a,env),nth(b,env),nth(z,env))

by (*simp add*: *is-cons-fm-def is-cons-def*)

definition
funspace-succ-fm :: *i* **where**
funspace-succ-fm ==
Exists(*Exists*(*Exists*(*Exists*(*And*(*pair-fm*(*3*,*2*,*4*),*And*(*pair-fm*(*6*,*2*,*1*),
And(*is-cons-fm*(*1*,*3*,*0*),*upair-fm*(*0*,*0*,*5*))))))))

lemma *funspace-succ-fm-type* [*TC*] :
funspace-succ-fm ∈ *formula*
by (*simp add*: *funspace-succ-fm-def*)

lemma [*simp*] : *arity*(*funspace-succ-fm*) = *3*
by (*simp add*: *funspace-succ-fm-def pair-fm-def upair-fm-def is-cons-fm-def*
union-fm-def Un-commute nat-union-abs1)

lemma (**in** *forcing-data*) *funspace-succ-rep-intf* :
assumes
n∈*M*
shows
strong-replacement(##*M*,
λ*p z*. ∃ *f*∈*M*. ∃ *b*∈*M*. ∃ *nb*∈*M*. ∃ *cnbf*∈*M*.
pair(##*M*,*f*,*b*,*p*) & *pair*(##*M*,*n*,*b*,*nb*) & *is-cons*(##*M*,*nb*,*f*,*cnbf*)
&
upair(##*M*,*cnbf*,*cnbf*,*z*))
proof −
from *replacement-ax* **have**
(∀ *a*∈*M*. *strong-replacement*(##*M*,λ*x y*. *sats*(*M*,*funspace-succ-fm*,[*x*,*y*,*a*])))

by *simp*
then have
(∀ *n*∈*M*. *strong-replacement*(##*M*,
λ*p z*. ∃ *f*∈*M*. ∃ *b*∈*M*. ∃ *nb*∈*M*. ∃ *cnbf*∈*M*.
pair(##*M*,*f*,*b*,*p*) & *pair*(##*M*,*n*,*b*,*nb*) & *is-cons*(##*M*,*nb*,*f*,*cnbf*)
&
upair(##*M*,*cnbf*,*cnbf*,*z*)))
unfolding *funspace-succ-fm-def strong-replacement-def univalent-def* **by** (*simp*
del: *pair-abs*)
with ‹*n*∈*M*› **show** *?thesis* **by** *simp*
qed

lemmas (**in** *forcing-data*) *M-basic-sep-instances* =
inter-sep-intf diff-sep-intf cartprod-sep-intf
image-sep-intf converse-sep-intf restrict-sep-intf
pred-sep-intf memrel-sep-intf comp-sep-intf is-recfun-sep-intf

```
sublocale forcing-data ⊆ M-basic ##M
  apply (insert trans-M zero-in-M power-ax)
  apply (rule M-basic.intro,rule mtriv)
  apply (rule M-basic-axioms.intro)
  apply (insert M-basic-sep-instances funspace-succ-rep-intf)
  apply (simp-all)
done

end
theory Recursion-Thms imports ZF.WF begin



lemma fld-restrict-eq : a ∈ A ⟹ (r∩A*A)-``{a} = (r-``{a} ∩ A)
  by(force)

lemma fld-restrict-mono : relation(r) ⟹ A ⊆ B ⟹ r∩A*A ⊆ r∩B*B
  by(auto)

lemma fld-restrict-dom :
  assumes relation(r) domain(r) ⊆ A range(r)⊆ A
  shows r∩A*A = r
  proof (rule equalityI,blast,rule subsetI)
    { fix x
    assume xr: x ∈ r
    from xr assms have ∃ a b . x = <a,b> by (simp add: relation-def)
    then obtain a b where <a,b> ∈ r <a,b> ∈ r∩A*A x ∈ r∩A*A
      using assms xr
      by force
    then have x∈ r ∩ A*A by simp
  }
  then show x ∈ r ⟹ x∈ r∩A*A for x .
qed

definition tr-down :: [i,i] ⇒ i
  where tr-down(r,a) = (r^+)-``{a}

lemma tr-downD : x ∈ tr-down(r,a) ⟹ <x,a> ∈ r^+
  by (simp add: tr-down-def vimage-singleton-iff)

lemma pred-down : relation(r) ⟹ r-``{a} ⊆ tr-down(r,a)
 by(simp add: tr-down-def vimage-mono r-subset-trancl)

lemma tr-down-mono : relation(r) ⟹ x ∈ r-``{a} ⟹ tr-down(r,x) ⊆ tr-down(r,a)
  by(rule subsetI,simp add:tr-down-def,auto dest: underD,force simp add: underI
r-into-trancl trancl-trans)

lemma rest-eq :
  assumes relation(r) and r-``{a} ⊆ B and a ∈ B
```

```
  shows r-``{a} = (r∩B*B)-``{a}
proof
  { fix x
    assume x ∈ r-``{a}
    then have x ∈ B using assms by (simp add: subsetD)
    from ‹x∈ r-``{a}› underD have <x,a> ∈ r by simp
    then have x ∈ (r∩B*B)-``{a} using ‹x ∈ B› ‹a∈B› underI by simp
  }
  then show r -`` {a} ⊆ (r∩B*B)-`` {a} by auto
next
  from vimage-mono assms
  show (r∩B*B) -`` {a} ⊆ r -`` {a} by auto
qed

lemma wfrec-restr-eq : r' = r ∩ A*A ⟹ wfrec[A](r,a,H) = wfrec(r',a,H)
  by(simp add:wfrec-on-def)

lemma wfrec-restr :
  assumes rr: relation(r) and wfr:wf(r)
  shows  a ∈ A ⟹ tr-down(r,a) ⊆ A ⟹ wfrec(r,a,H) = wfrec[A](r,a,H)
proof (induct a arbitrary:A rule:wf-induct-raw[OF wfr] )
  case (1 a)
  from wf-subset wfr wf-on-def Int-lower1 have wfRa : wf[A](r) by simp
  from pred-down rr have r -`` {a} ⊆ tr-down(r, a)  .
  then have r-``{a} ⊆ A using 1 by (force simp add: subset-trans)
  {
    fix x
    assume x-a : x ∈ r-``{a}
    with ‹r-``{a} ⊆ A› have x ∈ A ..
    from pred-down rr have b : r -``{x} ⊆ tr-down(r,x) .
    then have tr-down(r,x) ⊆ tr-down(r,a)
      using tr-down-mono x-a rr by simp
    then have tr-down(r,x) ⊆ A using 1 subset-trans by force
    have <x,a> ∈ r using x-a  underD by simp
    then have wfrec(r,x,H) = wfrec[A](r,x,H)
      using 1 ‹tr-down(r,x) ⊆ A› ‹x ∈ A› by simp
  }
  then have x∈ r-``{a} ⟹ wfrec(r,x,H) =  wfrec[A](r,x,H) for x  .
 then have Eq1 :(λ x ∈ r-``{a} . wfrec(r,x,H)) = (λ x ∈ r-``{a} . wfrec[A](r,x,H))

    using lam-cong by simp

  from assms have
    wfrec(r,a,H) = H(a,λ x ∈ r-``{a} . wfrec(r,x,H)) by (simp add:wfrec)
  also have ... = H(a,λ x ∈ r-``{a} . wfrec[A](r,x,H))
    using assms Eq1 by simp
  also have ... = H(a,λ x ∈ (r∩A*A)-``{a} . wfrec[A](r,x,H))
    using 1 assms rest-eq ‹r-``{a} ⊆ A› by simp
  also have ... = H(a,λ x ∈ (r-``{a})∩A . wfrec[A](r,x,H))
```

using ⟨$a \in A$⟩ *fld-restrict-eq* **by** *simp*
also have ... = $wfrec[A](r,a,H)$ **using** ⟨$wf[A](r)$⟩ ⟨$a \in A$⟩ *wfrec-on* **by** *simp*
finally show *?case* **.**
qed

lemmas *wfrec-tr-down* = *wfrec-restr*[*OF* - - - *subset-refl*]

lemma *wfrec-trans-restr* : $relation(r) \Longrightarrow wf(r) \Longrightarrow trans(r) \Longrightarrow r-``\{a\} \subseteq A \Longrightarrow a \in A \Longrightarrow$
$wfrec(r, a, H) = wfrec[A](r, a, H)$
by(*subgoal-tac* $tr\text{-}down(r,a) \subseteq A$,*auto simp add* : *wfrec-restr tr-down-def trancl-eq-r*)

end
theory *Names* **imports** *Forcing-Data Interface Recursion-Thms* **begin**

lemma *transD* : $Transset(M) \Longrightarrow y \in M \Longrightarrow y \subseteq M$
by (*unfold Transset-def*, *blast*)

definition
SepReplace :: $[i, i \Rightarrow i, i \Rightarrow o] \Rightarrow i$ **where**
$SepReplace(A,b,Q) == \{y \;.\; x \in A,\; y = b(x) \wedge Q(x)\}$

syntax
-SepReplace :: [*i*, *pttrn*, *i*, *o*] => *i* ((*1*{- ../ - ∈ -, -}))
translations
{*b* .. *x*∈*A*, *Q*} => *CONST SepReplace*(*A*, λ*x*. *b*, λ*x*. *Q*)

lemma *Sep-and-Replace*: $\{b(x) \;..\; x \in A,\; P(x)\} = \{b(x) \;.\; x \in \{y \in A.\; P(y)\}\}$
by (*auto simp add*:*SepReplace-def*)

lemma *SepReplace-subset* : $A \subseteq A' \Longrightarrow \{b \;..\; x \in A,\; Q\} \subseteq \{b \;..\; x \in A',\; Q\}$
by (*auto simp add*:*SepReplace-def*)

lemma *SepReplace-iff* [*simp*]: $y \in \{b(x) \;..\; x \in A,\; P(x)\} \longleftrightarrow (\exists x \in A.\; y = b(x) \;\&\; P(x))$
by (*auto simp add*:*SepReplace-def*)

lemma *SepReplace-dom-implies* :
$(\bigwedge x \;.\; x \in A \Longrightarrow b(x) = b'(x)) \Longrightarrow \{b(x) \;..\; x \in A,\; Q(x)\} = \{b'(x) \;..\; x \in A,\; Q(x)\}$
by (*simp add*:*SepReplace-def*)

lemma *SepReplace-pred-implies* :
$\forall x.\; Q(x) \longrightarrow b(x) = b'(x) \Longrightarrow \{b(x) \;..\; x \in A,\; Q(x)\} = \{b'(x) \;..\; x \in A,\; Q(x)\}$
by (*force simp add*:*SepReplace-def*)

11 eclose properties

lemma *eclose-sing* : x ∈ eclose(a) ⟹ x ∈ eclose({a})
by(*rule subsetD*[*OF mem-eclose-subset*],*simp*+)

lemma *ecloseE* : x ∈ eclose(A) ⟹ x ∈ A ∨ (∃ B ∈ A . x ∈ eclose(B))
apply(*erule eclose-induct-down*,*simp*,*erule disjE*,*rule disjI2*,*simp add*:*arg-into-eclose*)
apply(*subgoal-tac* z ∈ eclose(y),*blast*,*simp add*: *arg-into-eclose*)
apply(*rule disjI2*,*erule bexE*,*subgoal-tac* z ∈ eclose(B),*blast*,*simp add*:*ecloseD*)
done

lemma *eclose-singE* : x ∈ eclose({a}) ⟹ x = a ∨ x ∈ eclose(a)
by(*blast dest*: *ecloseE*)

lemma *in-eclose-sing* : x ∈ eclose({a}) ⟹ a ∈ eclose(z) ⟹ x ∈ eclose({z})
apply(*drule eclose-singE*,*erule disjE*,*simp add*: *eclose-sing*)
apply(*rule eclose-sing*,*erule mem-eclose-trans*,*assumption*)
done

lemma *in-dom-in-eclose* : x ∈ domain(z) ⟹ x ∈ eclose(z)
apply(*auto simp add*:*domain-def*)
apply(*rule-tac* A={x} **in** *ecloseD*)
apply(*subst* (*asm*) *Pair-def*)
apply(*rule-tac* A={{x,x},{x,y}} **in** *ecloseD*,*auto simp add*:*arg-into-eclose*)
done

The well founded relation on which *val* is defined

definition
ed :: [i,i] ⇒ o **where**
ed(x,y) == x ∈ domain(y)

definition
edrel :: i ⇒ i **where**
edrel(A) == {<x,y> ∈ A*A . x ∈ domain(y)}

lemma *edrel-dest* [*dest*]: x ∈ edrel(A) ⟹ ∃ a∈ A. ∃ b ∈ A. x =<a,b>
by(*auto simp add*:*edrel-def*)

lemma *edrelD* : x ∈ edrel(A) ⟹ ∃ a∈ A. ∃ b ∈ A. x =<a,b> ∧ a ∈ domain(b)
by(*auto simp add*:*edrel-def*)

lemma *edrelI* [*intro*!]: x∈A ⟹ y∈A ⟹ x ∈ domain(y) ⟹ <x,y>∈edrel(A)
by (*simp add*:*edrel-def*)

lemma *edrel-trans*: Transset(A) ⟹ y∈A ⟹ x ∈ domain(y) ⟹ <x,y>∈edrel(A)
by (*rule edrelI*, *auto simp add*:*Transset-def domain-def Pair-def*)

lemma *domain-trans*: Transset(A) ⟹ y∈A ⟹ x ∈ domain(y) ⟹ x∈A
by (*auto simp add*: *Transset-def domain-def Pair-def*)

lemma *relation-edrel* : *relation*(*edrel*(*A*))
  by(*auto simp add*: *relation-def*)

lemma *edrel-sub-memrel*: *edrel*(*A*) ⊆ *trancl*(*Memrel*(*eclose*(*A*)))
proof
  fix *z*
  assume
    z∈*edrel*(*A*)
  then obtain *x y* **where**
    Eq1:   *x*∈*A* *y*∈*A* *z*=<*x*,*y*> *x*∈*domain*(*y*)
    by (*auto simp add*: *edrel-def*)
  then obtain *u v* **where**
    Eq2:   *x*∈*u* *u*∈*v* *v*∈*y*
    unfolding *domain-def Pair-def* **by** *auto*
  with *Eq1* **have**
    Eq3:   *x*∈*eclose*(*A*) *y*∈*eclose*(*A*) *u*∈*eclose*(*A*) *v*∈*eclose*(*A*)
   by (*auto*, *rule-tac* [*3*−*4*] *ecloseD*, *rule-tac* [*3*] *ecloseD*, *simp-all add*:*arg-into-eclose*)
  let
    ?r=*trancl*(*Memrel*(*eclose*(*A*)))
  from *Eq2* **and** *Eq3* **have**
    <*x*,*u*>∈*?r* <*u*,*v*>∈*?r* <*v*,*y*>∈*?r*
    by (*auto simp add*: *r-into-trancl*)
  then have
    <*x*,*y*>∈*?r*
    by (*rule-tac trancl-trans*, *rule-tac* [*2*] *trancl-trans*, *simp*)
  with *Eq1* **show** *z*∈*?r* **by** *simp*
qed

lemma *wf-edrel* : *wf*(*edrel*(*A*))
  apply (*rule-tac wf-subset* [*of trancl*(*Memrel*(*eclose*(*A*)))])
   apply (*auto simp add*:*edrel-sub-memrel wf-trancl wf-Memrel*)
  done

lemma *dom-under-edrel-eclose*: *edrel*(*eclose*({*x*})) −ʻʻ {*x*}= *domain*(*x*)
  apply(*simp add*:*edrel-def*,*rule*,*rule*,*drule underD*,*simp*,*rule*,*rule underI*)
  apply(*auto simp add*:*in-dom-in-eclose eclose-sing arg-into-eclose*)
  done

lemma *ed-eclose* : <*y*,*z*> ∈ *edrel*(*A*) ⟹ *y* ∈ *eclose*(*z*)
  by(*drule edrelD*,*auto simp add*:*domain-def in-dom-in-eclose*)

lemma *tr-edrel-eclose* : <*y*,*z*> ∈ *edrel*(*eclose*({*x*}))ˆ+ ⟹ *y* ∈ *eclose*(*z*)
  by(*rule trancl-induct*,(*simp add*: *ed-eclose mem-eclose-trans*)+)

```
lemma restrict-edrel-eq :
  assumes z ∈ domain(x)
  shows edrel(eclose({x}))∩ eclose({z})∗eclose({z}) = edrel(eclose({z}))
proof
  let ?ec=λ y . edrel(eclose({y}))
  let ?ez=eclose({z})
  let ?rr=?ec(x)∩?ez∗?ez
  { fix y
    assume yr:y ∈ ?rr
    with yr obtain a b where 1:<a,b> ∈ ?rr∩?ez∗?ez
      a ∈ ?ez b ∈ ?ez <a,b> ∈ ?ec(x) y=<a,b> by blast
    then have a ∈ domain(b) using edrelD by blast
    with 1 have y ∈ edrel(eclose({z})) by blast
  }
  then show ?rr ⊆ edrel(?ez) using subsetI by auto
next
  let ?ec=λ y . edrel(eclose({y}))
  let ?ez=eclose({z})
  let ?rr=?ec(x)∩?ez∗?ez
  { fix y
    assume yr:y ∈ edrel(?ez)
    then obtain a b where 1: a ∈ ?ez b ∈ ?ez y=<a,b> a ∈ domain(b)
      using edrelD by blast
    with assms have z ∈ eclose(x) using in-dom-in-eclose by simp
     with assms 1 have a ∈ eclose({x}) b ∈ eclose({x}) using in-eclose-sing by
simp-all
    with ‹a∈domain(b)› have <a,b> ∈ edrel(eclose({x})) by blast
    with 1 have y ∈ ?rr by simp
  }
  then show edrel(eclose({z})) ⊆ ?rr by blast
qed

lemma tr-edrel-subset :
  assumes z ∈ domain(x)
  shows   tr-down(edrel(eclose({x})),z) ⊆ eclose({z})
proof −
  let ?r=λ x . edrel(eclose({x}))
  {fix y
    assume  y ∈ tr-down(?r(x),z)
    then have <y,z> ∈ ?r(x)^+ using tr-downD by simp
    with assms have y ∈ eclose({z}) using tr-edrel-eclose eclose-sing by simp
  }
  then show ?thesis  by blast
qed


context forcing-data
begin
```

```
lemma upairM : x ∈ M ⟹ y ∈ M ⟹ {x,y} ∈ M
 by (simp del:setclass-iff  add:setclass-iff [symmetric])

lemma singletonM : a ∈ M ⟹ {a} ∈ M
 by (simp del:setclass-iff  add:setclass-iff [symmetric])

lemma pairM : x ∈  M ⟹ y ∈ M ⟹ <x,y> ∈ M
 by (simp del:setclass-iff  add:setclass-iff [symmetric])

lemma P-sub-M : P ⊆ M
  by (simp add: P-in-M trans-M transD)

lemma Rep-simp : Replace(u,λ y z . z = f(y)) = { f(y) . y ∈ u}
  by(auto)

definition
  Hcheck :: [i,i] ⇒ i where
  Hcheck(z,f)  == { <f`y,one> . y ∈ z}

definition
  check :: i ⇒ i where
  check(x) == transrec(x , Hcheck)

lemma checkD:
  check(x) =  wfrec(Memrel(eclose({x})), x, Hcheck)
  unfolding check-def transrec-def ..

lemma  aux-def-check: x ∈ y ⟹
  wfrec(Memrel(eclose({y})), x, Hcheck) =
  wfrec(Memrel(eclose({x})), x, Hcheck)
  by (rule wfrec-eclose-eq,auto simp add: arg-into-eclose eclose-sing)

lemma def-check : check(y) = { <check(w),one> . w ∈ y}
proof −
  let
    ?r=λy. Memrel(eclose({y}))
  from wf-Memrel have
    wfr:  ∀ w . wf(?r(w)) ..
  with wfrec [of ?r(y) y Hcheck] have
    check(y)= Hcheck( y, λx∈?r(y) −`` {y}. wfrec(?r(y), x, Hcheck))
    using checkD by simp
  also have
     ... = Hcheck( y, λx∈y. wfrec(?r(y), x, Hcheck))
    using under-Memrel-eclose arg-into-eclose by simp
  also have
     ... = Hcheck( y, λx∈y. check(x))
    using aux-def-check checkD by simp
  finally show ?thesis using Hcheck-def by simp
qed
```

```
lemma def-checkS :
  fixes n
  assumes n ∈ nat
  shows check(succ(n)) = check(n) ∪ {<check(n),one>}
proof −
  have check(succ(n)) = {<check(i),one> . i ∈ succ(n)}
    using def-check by blast
  also have ... = {<check(i),one> . i ∈ n} ∪ {<check(n),one>}
    by blast
  also have ... = check(n) ∪ {<check(n),one>}
    using def-check[of n,symmetric] by simp
  finally show ?thesis .
qed

lemma field-Memrel : x ∈ M ⟹ field(Memrel(eclose({x}))) ⊆ M
  apply(rule subset-trans,rule field-rel-subset,rule Ordinal.Memrel-type)
  apply(rule eclose-least,rule trans-M,auto)
  done

definition
  Hv :: i⇒i⇒i⇒i where
  Hv(G,x,f) == { f‘y .. y∈ domain(x), ∃ p∈P. <y,p> ∈ x ∧ p ∈ G }

definition
  val :: i⇒i⇒i where
  val(G,τ) == wfrec(edrel(eclose({τ})), τ ,Hv(G))

lemma aux-def-val:
  assumes z ∈ domain(x)
  shows wfrec(edrel(eclose({x})),z,Hv(G)) = wfrec(edrel(eclose({z})),z,Hv(G))
proof −
  let ?r=λx . edrel(eclose({x}))
  have z∈eclose({z}) using arg-in-eclose-sing .
  moreover have relation(?r(x)) using relation-edrel .
  moreover have wf(?r(x)) using wf-edrel .
 moreover from assms have tr-down(?r(x),z) ⊆ eclose({z}) using tr-edrel-subset
by simp
  ultimately have
    wfrec(?r(x),z,Hv(G)) = wfrec[eclose({z})](?r(x),z,Hv(G))
    using wfrec-restr by simp
  also from ‹z∈domain(x)› have ... = wfrec(?r(z),z,Hv(G))
      using restrict-edrel-eq wfrec-restr-eq by simp
  finally show ?thesis .
qed

lemma def-val:  val(G,x) = {val(G,t) .. t∈domain(x) , ∃ p∈P . <t,p>∈x ∧ p ∈
G }
```

proof –
let
?r=λτ . edrel(eclose({τ}))
let
?f=λz∈?r(x)–``{x}. wfrec(?r(x),z,Hv(G))
have ∀τ. wf(?r(τ)) **using** wf-edrel **by** simp
with wfrec [of - x] **have**
val(G,x) = Hv(G,x,?f) **using** val-def **by** simp
also have
... = Hv(G,x,λz∈domain(x). wfrec(?r(x),z,Hv(G)))
using dom-under-edrel-eclose **by** simp
also have
... = Hv(G,x,λz∈domain(x). val(G,z))
using aux-def-val val-def **by** simp
finally show ?thesis **using** Hv-def SepReplace-def **by** simp
qed

lemma val-mono : x⊆y ⟹ val(G,x) ⊆ val(G,y)
by (subst (1 2) def-val, force)

lemma valcheck : one ∈ G ⟹ one ∈ P ⟹ val(G,check(y)) = y
proof (induct rule:eps-induct)
case (1 y)
then show ?case
proof –
from def-check **have**
Eq1: check(y) = { ⟨check(w), one⟩ . w ∈ y} (**is** - = ?C) **.**
from Eq1 **have**
val(G,check(y)) = val(G, {⟨check(w), one⟩ . w ∈ y})
by simp
also have
... = {val(G,t) .. t∈domain(?C) , ∃p∈P . ⟨t, p⟩∈?C ∧ p ∈ G }
using def-val **by** blast
also have
... = {val(G,t) .. t∈domain(?C) , ∃w∈y. t=check(w) }
using 1 **by** simp
also have
... = {val(G,check(w)) . w∈y }
by force
finally show
val(G,check(y)) = y
using 1 **by** simp
qed
qed

lemma val-of-name :
val(G,{x∈A×P. Q(x)}) = {val(G,t) .. t∈A , ∃p∈P . Q(<t,p>) ∧ p ∈ G }
proof –
let

```
    ?n={x∈A×P. Q(x)} and
    ?r=λτ . edrel(eclose({τ}))
  let
    ?f=λz∈?r(?n)−‘‘{?n}. val(G,z)
  have
    wfR : wf(?r(τ)) for τ
    by (simp add: wf-edrel)
  have domain(?n) ⊆ A by auto
  { fix t
    assume H:t ∈ domain({x ∈ A × P . Q(x)})
    then have ?f ‘ t = (if t ∈ ?r(?n)−‘‘{?n} then val(G,t) else 0)
      by simp
    moreover have ... = val(G,t)
      using dom-under-edrel-eclose H if-P by auto
  }
  then have Eq1: t ∈ domain({x ∈ A × P . Q(x)}) =⇒
    val(G,t) = ?f‘ t  for t
    by simp
  have
    val(G,?n) = {val(G,t) .. t∈domain(?n), ∃ p ∈ P . <t,p> ∈ ?n ∧ p ∈ G}
    by (subst def-val,simp)
  also have
    ... = {?f‘t .. t∈domain(?n), ∃ p∈P . <t,p>∈?n ∧ p∈G}
    unfolding Hv-def
    by (subst SepReplace-dom-implies,auto simp add:Eq1)
  also have
     ... = { (if t∈?r(?n)−‘‘{?n} then val(G,t) else 0) .. t∈domain(?n), ∃ p∈P .
<t,p>∈?n ∧ p∈G}
    by (simp)
  also have
    Eq2:  ... = { val(G,t) .. t∈domain(?n), ∃ p∈P . <t,p>∈?n ∧ p∈G}
  proof −
    from dom-under-edrel-eclose have
      domain(?n) ⊆ ?r(?n)−‘‘{?n}
      by simp
    then have
      ∀ t∈domain(?n). (if t∈?r(?n)−‘‘{?n} then val(G,t) else 0) = val(G,t)
      by auto
    then show
        { (if t∈?r(?n)−‘‘{?n} then val(G,t) else 0) .. t∈domain(?n), ∃ p∈P .
<t,p>∈?n ∧ p∈G} =
            { val(G,t) .. t∈domain(?n), ∃ p∈P . <t,p>∈?n ∧ p∈G}
      by auto
  qed
  also have
     ... = { val(G,t) .. t∈A, ∃ p∈P . <t,p>∈?n ∧ p∈G}
    by force
  finally show
     val(G,?n)  = { val(G,t) .. t∈A, ∃ p∈P . Q(<t,p>) ∧ p∈G}
```

by *auto*
qed

lemma *val-of-name-alt* :
val(*G*,{*x*∈*A*×*P*. *Q*(*x*)}) = {*val*(*G*,*t*) .. *t*∈*A* , ∃ *p*∈*P*∩*G* . *Q*(<*t*,*p*>) }
using *val-of-name* **by** *force*

definition
GenExt :: *i*⇒*i* (*M*[-])
where *GenExt*(*G*)== {*val*(*G*,τ). τ ∈ *M*}

lemma *val-of-elem*: <ϑ,*p*> ∈ π ⟹ *p*∈*G* ⟹ *p*∈*P* ⟹ *val*(*G*,ϑ) ∈ *val*(*G*,π)
proof −
assume
<ϑ,*p*> ∈ π
then have ϑ∈*domain*(π) **by** *auto*
assume
p∈*G* *p*∈*P*
with ‹ϑ∈*domain*(π)› ‹<ϑ,*p*> ∈ π› **have**
val(*G*,ϑ) ∈ {*val*(*G*,*t*) .. *t*∈*domain*(π) , ∃ *p*∈*P* . ⟨*t*, *p*⟩∈π ∧ *p* ∈ *G* }
by *auto*
then show *?thesis* **by** (*subst def-val*)
qed

lemma *elem-of-val*: *x*∈*val*(*G*,π) ⟹ ∃ ϑ∈*domain*(π). *val*(*G*,ϑ) = *x*
by (*subst* (*asm*) *def-val*,*auto*)

lemma *elem-of-val-pair*: *x*∈*val*(*G*,π) ⟹ ∃ ϑ. ∃ *p*∈*G*. <ϑ,*p*>∈π ∧ *val*(*G*,ϑ) = *x*
by (*subst* (*asm*) *def-val*,*auto*)

lemma *GenExtD*:
x ∈ *M*[*G*] ⟹ ∃ τ∈*M*. *x* = *val*(*G*,τ)
by (*simp add*:*GenExt-def*)

lemma *GenExtI*:
x ∈ *M* ⟹ *val*(*G*,*x*) ∈ *M*[*G*]
by (*auto simp add*: *GenExt-def*)

lemma *Transset-MG* : *Transset*(*M*[*G*])
proof −
{ **fix** *vc y*
assume *vc* ∈ *M*[*G*] **and** *y* ∈ *vc*
from ‹*vc*∈*M*[*G*]› **and** ‹*y* ∈ *vc*› **obtain** *c* **where**
c∈*M* *val*(*G*,*c*)∈*M*[*G*] *y* ∈ *val*(*G*,*c*)
using *GenExtD* **by** *auto*
from ‹*y* ∈ *val*(*G*,*c*)› **obtain** ϑ **where**
ϑ∈*domain*(*c*) *val*(*G*,ϑ) = *y* **using** *elem-of-val* **by** *blast*
with *trans-M* ‹*c*∈*M*›

have *y* ∈ *M*[*G*] **using** *domain-trans GenExtI* **by** *blast*
}
then show *?thesis* **using** *Transset-def* **by** *auto*
qed

lemma *check-n-M* :
fixes *n*
assumes *n* ∈ *nat*
shows *check*(*n*) ∈ *M*
using ‹*n*∈*nat*› **proof** (*induct n*)
case *0*
then show *?case* **using** *zero-in-M* **by** (*subst def-check*,*simp*)
next
case (*succ x*)
have *one* ∈ *M* **using** *one-in-P P-sub-M subsetD* **by** *simp*
with ‹*check*(*x*)∈*M*› **have** <*check*(*x*),*one*> ∈ *M* **using** *pairM* **by** *simp*
then have {<*check*(*x*),*one*>} ∈ *M* **using** *singletonM* **by** *simp*
with ‹*check*(*x*)∈*M*› **have** *check*(*x*) ∪ {<*check*(*x*),*one*>} ∈ *M* **using** *Un-closed* **by** *simp*
then show *?case* **using** ‹*x*∈*nat*› *def-checkS* **by** *simp*
qed

end

locale *M-extra-assms* = *forcing-data* +
assumes
check-in-M : ⋀*x*. *x* ∈ *M* ⟹ *check*(*x*) ∈ *M*
and *repl-check-pair* : *strong-replacement*(##*M*,λ*p y*. *y* =<*check*(*p*),*p*>)

begin
definition
G-dot :: *i* **where**
G-dot == {<*check*(*p*),*p*> . *p*∈*P*}

lemma *G-dot-in-M* :
G-dot ∈ *M*
proof −
have *0*:*G-dot* = { *y* . *p*∈*P* , *y* = <*check*(*p*),*p*> }
unfolding *G-dot-def* **by** *auto*
from *P-in-M check-in-M pairM P-sub-M* **have**
1: *p*∈*P* ⟹ <*check*(*p*),*p*> ∈ *M* **for** *p*
by *auto*
with *1 repl-check-pair P-in-M strong-replacement-closed* **have**
{ *y* . *p*∈*P* , *y* = <*check*(*p*),*p*> } ∈ *M* **by** *simp*
then show *?thesis* **using** *0* **by** *simp*
qed

lemma *val-G-dot* :

```
  assumes G ⊆ P
    one ∈ G
  shows val(G,G-dot) = G
proof (intro equalityI subsetI)
  fix x
  assume x∈val(G,G-dot)
  then obtain ϑ p where
    p∈G <ϑ,p> ∈ G-dot val(G,ϑ) = x ϑ = check(p)
    unfolding G-dot-def using elem-of-val-pair G-dot-in-M
    by force
  with ‹one∈G› ‹G⊆P› show
      x ∈ G
    using valcheck P-sub-M by auto
next
  fix p
  assume p∈G
  have q∈P ⟹ <check(q),q> ∈ G-dot for q
    unfolding G-dot-def by simp
  with ‹p∈G› ‹G⊆P› have
    val(G,check(p)) ∈ val(G,G-dot)
    using val-of-elem G-dot-in-M by blast
  with ‹p∈G› ‹G⊆P› ‹one∈G› show
    p ∈ val(G,G-dot)
    using P-sub-M valcheck by auto
qed


lemma G-in-Gen-Ext :
  assumes G ⊆ P and one ∈ G
  shows   G ∈ M[G]
 using assms val-G-dot GenExtI[of - G] G-dot-in-M
  by force

end

end
theory Extensionality-Axiom
imports
  Names
begin

context forcing-data
begin

lemma extensionality-in-MG : extensionality(##(M[G]))
proof –
  {
    fix x y z
    assume
```

```
      asms: x∈M[G] y∈M[G] (∀ w∈M[G] . w ∈ x ⟷ w ∈ y)
    from ‹x∈M[G]› have
      z∈x ⟷ z∈M[G] ∧ z∈x
      using Transset-MG Transset-intf by auto
    also have
      ... ⟷ z∈y
      using asms Transset-MG Transset-intf by auto
    finally have
      z∈x ⟷ z∈y .
  }
  then have
    ∀ x∈M[G] . ∀ y∈M[G] . (∀ z∈M[G] . z ∈ x ⟷ z ∈ y) ⟶ x = y
    by blast
  then show ?thesis unfolding extensionality-def by simp
qed

end
end
theory Foundation-Axiom
imports
  Names
begin

context forcing-data
begin


lemma foundation-in-MG : foundation-ax(##(M[G]))
  unfolding foundation-ax-def
 by (rule rallI, cut-tac A=x in foundation, auto intro: Transset-M [OF Transset-MG])


lemma foundation-ax(##(M[G]))
proof −
  {
    fix x
    assume
      x∈M[G] ∃ y∈M[G] . y∈x
    then have
      ∃ y∈M[G] . y∈x∩M[G]
      by simp
    then obtain y where
      y∈x∩M[G] ∀ z∈y. z ∉ x∩M[G]
      using foundation[of x∩M[G]]  by blast
    then have
      ∃ y∈M[G] . y ∈ x ∧ (∀ z∈M[G] . z ∉ x ∨ z ∉ y)
      by auto
  }
  then show ?thesis
```

unfolding *foundation-ax-def* **by** *auto*
qed

end
end
theory *Forcing-Theorems* **imports** *Interface Names* **begin**

locale *forcing-thms* = *forcing-data* +
fixes *forces* :: *i* ⇒ *i*
assumes *definition-of-forces*: *p*∈*P* ⟹ *φ*∈*formula* ⟹ *env*∈*list*(*M*) ⟹
sats(*M*,*forces*(*φ*), [*P*,*leq*,*one*,*p*] @ *env*) ⟷
(∀ *G*.(*M-generic*(*G*)∧ *p*∈*G*)⟶*sats*(*M*[*G*],*φ*,*map*(*val*(*G*),*env*)))
and *definability*[*TC*]: *φ*∈*formula* ⟹ *forces*(*φ*) ∈ *formula*
and *arity-forces*: *φ*∈*formula* ⟹ *arity*(*forces*(*φ*)) = *arity*(*φ*) #+ *4*
and *truth-lemma*: *φ*∈*formula* ⟹ *env*∈*list*(*M*) ⟹ *M-generic*(*G*) ⟹
(∃ *p*∈*G*.(*sats*(*M*,*forces*(*φ*), [*P*,*leq*,*one*,*p*] @ *env*))) ⟷
sats(*M*[*G*],*φ*,*map*(*val*(*G*),*env*))
and *strengthening*: *p*∈*P* ⟹ *φ*∈*formula* ⟹ *env*∈*list*(*M*) ⟹ *q*∈*P* ⟹
<*q*,*p*>∈*leq* ⟹
sats(*M*,*forces*(*φ*), [*P*,*leq*,*one*,*p*] @ *env*) ⟹
sats(*M*,*forces*(*φ*), [*P*,*leq*,*one*,*q*] @ *env*)
and *density-lemma*: *p*∈*P* ⟹ *φ*∈*formula* ⟹ *env*∈*list*(*M*) ⟹
sats(*M*,*forces*(*φ*), [*P*,*leq*,*one*,*p*] @ *env*) ⟷
dense-below({*q*∈*P*. *sats*(*M*,*forces*(*φ*), [*P*,*leq*,*one*,*q*] @ *env*)},*p*)

begin
end

locale *G-generic* = *forcing-thms* +
fixes *G* :: *i*
assumes *generic* : *M-generic*(*G*)
begin

lemma *zero-in-MG* :
0 ∈ *M*[*G*]
proof −
from *zero-in-M* **and** *elem-of-val* **have**
0 = *val*(*G*,*0*)
by *auto*
also from *GenExtI* **and** *zero-in-M* **have**
... ∈ *M*[*G*]
by *simp*
finally show *?thesis* **.**
qed

lemma *G-nonempty*: *G*≠*0*
proof −
have *P*⊆*P* **..**

```
  with P-in-M P-dense ‹P⊆P› show
    G ≠ 0
    using generic unfolding M-generic-def by auto
qed

end

end
theory Separation-Axiom
  imports Forcing-Theorems Renaming
begin

definition
  perm-sep-forces :: i where
 perm-sep-forces == {<0,3>,<1,4>,<2,5>,<3,1>,<4,0>,<5,6>,<6,7>,<7,2>}


lemma perm-sep-ftc : perm-sep-forces ∈ 8 −||> 8
  by(unfold perm-sep-forces-def,(rule consI,auto)+,rule emptyI)

lemma dom-perm-sep : domain(perm-sep-forces) = 8
  by(unfold perm-sep-forces-def,auto)

lemma perm-sep-tc : perm-sep-forces ∈ 8 → 8
  by(subst dom-perm-sep[symmetric],rule FiniteFun-is-fun,rule perm-sep-ftc)

lemma perm-sep-env :
  {p,q,r,s,t,u,v,w} ⊆ A ⟹ j<8 ⟹
  nth(j,[t,s,w,p,q,r,u,v]) = nth(perm-sep-forces‘j,[q,p,v,t,s,w,r,u])
  apply(subgoal-tac j∈nat)
 apply(rule natE,simp,subst apply-fun,rule perm-sep-tc,simp add:perm-sep-forces-def,simp-all)+
 apply(subst apply-fun,rule perm-sep-tc,simp add:perm-sep-forces-def,simp-all,drule
ltD,auto)
  done

context G-generic begin

lemmas transitivity = Transset-intf trans-M

lemma one-in-M: one ∈ M
  by (insert one-in-P P-in-M, simp add: transitivity)

lemma six-sep-aux:
  assumes
    b ∈ M [σ, π] ∈ list(M) ψ∈formula arity(ψ) ≤ 6
  shows
    {u ∈ b. sats(M,ψ,[u] @ [P, leq, one] @ [σ, π])} ∈ M
proof −
  from assms P-in-M leq-in-M one-in-M have
```

($\forall$ *u*$\in$*M*. *separation*(##*M*,λ*x*. *sats*(*M*,ψ,[*x*] @ [*P*, *leq*, *one*] @ [σ, π])))
using *sixp-sep* **by** *simp*
with ‹*b* $\in$ *M*› **show** *?thesis*
using *separation-iff* **by** *auto*
qed

lemma *Collect-sats-in-MG* :
assumes
$\pi \in M$ $\sigma \in M$ $val(G, \pi) = c$ $val(G, \sigma) = w$
$\varphi \in formula$ $arity(\varphi) \leq 2$
shows
$\{x \in c.\ sats(M[G], \varphi, [x, w])\} \in M[G]$
proof –
let
?χ=*And*(*Member*(*0*,*2*),φ)
and
?Pl1=[*P*,*leq*,*one*]
let
?new-form=*ren*(*forces*(*?*χ))‘*8*‘*8*‘*perm-sep-forces*
let
?ψ=*Exists*(*Exists*(*And*(*pair-fm*(*0*,*1*,*2*),*?new-form*)))
have *8*$\in$*nat* **by** *simp*
note *phi* = ‹$\varphi$$\in$*formula*› ‹*arity*($\varphi$) $\leq$ *2*›
then have
arity(*?*χ) $\leq$ *3*
using *nat-simp-union leI* **by** *simp*
with *phi* **have**
arity(*forces*(*?*χ)) $\leq$ *8*
using *nat-simp-union arity-forces leI* **by** *simp*
with *phi definability*[*of ?*χ] *arity-forces* **have**
?new-form $\in$ *formula*
using *ren-tc*[*of forces*(*?*χ) *8 8 perm-sep-forces*] *perm-sep-tc*
by *simp*
then have
?ψ $\in$ *formula*
by *simp*
from ‹$\varphi$$\in$*formula*› **have**
forces(*?*χ) $\in$ *formula*
using *definability* **by** *simp*
with ‹*arity*(*forces*(*?*χ)) $\leq$ *8*› **have**
arity(*?new-form*) $\leq$ *8*
using *ren-arity perm-sep-tc definability* **by** *simp*
then have
arity(*?*ψ) $\leq$ *6*
unfolding *pair-fm-def upair-fm-def*
using *nat-simp-union pred2-Un*[*of 8*] **by** *simp*
from ‹$\pi$$\in$*M*› ‹$\sigma$$\in$*M*› *P-in-M* **have**
domain(π)$\in$*M* *domain*(π) $\times$ *P* $\in$ *M*
by (*simp-all del*:*setclass-iff add*:*setclass-iff*[*symmetric*])

note *in-M* = ‹π∈M› ‹σ∈M› ‹domain(π) × P ∈ M› *P-in-M one-in-M leq-in-M*
{
fix *u*
assume
u ∈ *domain*(π) × *P u* ∈ *M*
with *in-M* ‹?new-form ∈ formula› ‹?ψ∈formula› **have**
Eq1: *sats*(*M*,?ψ,[*u*] @ ?*Pl1* @ [σ,π]) ⟷
(∃ϑ∈*M*. ∃*p*∈*P*. *u* =⟨ϑ,*p*⟩ ∧
sats(*M*,?*new-form*,[ϑ,*p*,*u*]@?*Pl1*@[σ,π]))
by (*auto simp add*: *transitivity*)
have
Eq3: ϑ∈*M* ⟹ *p*∈*P* ⟹
sats(*M*,?*new-form*,[ϑ,*p*,*u*]@?*Pl1*@[σ,π]) ⟷
(∀*F*. *M-generic*(*F*) ∧ *p* ∈ *F* ⟶ *sats*(*M*[*F*], ?χ, [*val*(*F*, ϑ), *val*(*F*, σ), *val*(*F*, π)]))
for ϑ *p*
proof −
fix *p* ϑ
assume
ϑ ∈ *M p*∈*P*
with *P-in-M* **have** *p*∈*M* **by** (*simp add*: *transitivity*)
note
in-M′ = *in-M* ‹ϑ ∈ M› ‹p∈M› ‹u ∈ domain(π) × P› ‹u ∈ M›
then have
[ϑ,σ,*u*] ∈ *list*(*M*) **by** *simp*
let
?*env*=?*Pl1*@[*p*,ϑ,σ,π,*u*]
let
?*new-env*= [ϑ,*p*,*u*,*P*,*leq*,*one*,σ,π]
let
?ψ=*Exists*(*Exists*(*And*(*pair-fm*(*0*,*1*,*2*),?*new-form*)))
have
?χ ∈ *formula arity*(?χ) ≤ *3 forces*(?χ)∈ *formula*
using *phi nat-simp-union leI* **by** *auto*
with *arity-forces* **have**
arity(*forces*(?χ)) ≤ *7*
by *simp*
then have *arity*(*forces*(?χ)) ≤ *8* **using** *le-trans* **by** *simp*
from *in-M′* **have**
?*Pl1* ∈ *list*(*M*) **by** *simp*
from *in-M′* **have** ?*env* ∈ *list*(*M*) **by** *simp*
have
Eq1′: ?*new-env* ∈ *list*(*M*) **using** *in-M′* **by** *simp*
then have
sats(*M*,?*new-form*,[ϑ,*p*,*u*]@?*Pl1*@[σ,π]) ⟷ *sats*(*M*,?*new-form*,?*new-env*)
by *simp*
also from ‹forces(?χ)∈formula› ‹8∈nat› ‹?env∈list(M)›
‹?new-env∈list(M)› *perm-sep-tc* ‹arity(forces(?χ)) ≤ 8›
have

... ⟷ *sats*(*M*,*forces*(*?χ*),*?env*)
using *sats-iff-sats-ren*[*of - 8 8 ?env M ?new-env*] *perm-sep-env*
by *auto*
also have
... ⟷ *sats*(*M*,*forces*(*?χ*), [*P*, *leq*, *one*,*p*,*ϑ*,*σ*,*π*]@[*u*]) **by** *simp*
also from ‹*arity*(*forces*(*?χ*)) ≤ *7*› ‹*forces*(*?χ*)∈*formula*› *in-M'* *phi* **have**
... ⟷ *sats*(*M*,*forces*(*?χ*), [*P*, *leq*, *one*,*p*,*ϑ*,*σ*,*π*])
by (*rule-tac arity-sats-iff* ,*auto*)
also from ‹*arity*(*forces*(*?χ*)) ≤ *7*› ‹*forces*(*?χ*)∈*formula*› *in-M'* *phi* **have**
... ⟷ (∀ *F*. *M-generic*(*F*) ∧ *p* ∈ *F* ⟶
sats(*M*[*F*], *?χ*, [*val*(*F*, *ϑ*), *val*(*F*, *σ*), *val*(*F*, *π*)]))
using *definition-of-forces*
proof (*intro iffI*)
assume
a1: *sats*(*M*, *forces*(*?χ*), [*P*, *leq*, *one*,*p*,*ϑ*,*σ*,*π*])
note *definition-of-forces*
then have
p ∈ *P* ⟹ *?χ*∈*formula* ⟹ [*ϑ*,*σ*,*π*] ∈ *list*(*M*) ⟹
sats(*M*, *forces*(*?χ*), [*P*, *leq*, *one*, *p*] @ [*ϑ*,*σ*,*π*]) ⟹
∀ *G*. *M-generic*(*G*) ∧ *p* ∈ *G* ⟶ *sats*(*M*[*G*], *?χ*, *map*(*val*(*G*), [*ϑ*,*σ*,*π*]))
..
then show
∀ *F*. *M-generic*(*F*) ∧ *p* ∈ *F* ⟶
sats(*M*[*F*], *?χ*, [*val*(*F*, *ϑ*), *val*(*F*, *σ*), *val*(*F*, *π*)])
using ‹*?χ*∈*formula*› ‹*p*∈*P*› *a1* ‹*ϑ*∈*M*› ‹*σ*∈*M*› ‹*π*∈*M*› **by** *auto*
next
assume
∀ *F*. *M-generic*(*F*) ∧ *p* ∈ *F* ⟶
sats(*M*[*F*], *?χ*, [*val*(*F*, *ϑ*), *val*(*F*, *σ*), *val*(*F*, *π*)])
with *definition-of-forces* [*THEN iffD2*] **show**
sats(*M*, *forces*(*?χ*), [*P*, *leq*, *one*,*p*,*ϑ*,*σ*,*π*])
using ‹*?χ*∈*formula*› ‹*p*∈*P*› *in-M'* **by** *auto*
qed
finally show
sats(*M*,*?new-form*,[*ϑ*,*p*,*u*]@*?Pl1*@[*σ*,*π*]) ⟷ (∀ *F*. *M-generic*(*F*) ∧ *p* ∈ *F*
⟶
sats(*M*[*F*], *?χ*, [*val*(*F*, *ϑ*), *val*(*F*, *σ*), *val*(*F*, *π*)])) **by** *simp*
qed
with *Eq1* **have**
sats(*M*,*?ψ*,[*u*] @ *?Pl1* @ [*σ*,*π*]) ⟷
(∃ *ϑ*∈*M*. ∃ *p*∈*P*. *u* =⟨*ϑ*,*p*⟩ ∧
(∀ *F*. *M-generic*(*F*) ∧ *p* ∈ *F* ⟶ *sats*(*M*[*F*], *?χ*, [*val*(*F*, *ϑ*), *val*(*F*, *σ*),
val(*F*, *π*)])))
by *auto*
}
then have
Equivalence: *u*∈ *domain*(*π*) × *P* ⟹ *u* ∈ *M* ⟹
sats(*M*,*?ψ*,[*u*] @ *?Pl1* @ [*σ*,*π*]) ⟷
(∃ *ϑ*∈*M*. ∃ *p*∈*P*. *u* =⟨*ϑ*,*p*⟩ ∧

($\forall F.$ *M-generic*$(F) \wedge p \in F \longrightarrow$ *sats*$(M[F],$ *?*$\chi,$ $[$*val*$(F, \vartheta),$ *val*$(F, \sigma),$ *val*$(F, \pi)])))$
for u
by *simp*
with *generic* **have**
$u \in$ *domain*$(\pi) \times P \Longrightarrow u \in M \Longrightarrow$
sats$(M,$*?*$\psi,[u, P,$ *leq*, *one*$,\sigma, \pi]) \Longrightarrow$
$(\exists \vartheta \in M.\ \exists p \in P.\ u = \langle \vartheta,p \rangle \wedge$
$(p \in G \longrightarrow$ *sats*$(M[G],$ *?*$\chi,$ $[$*val*$(G, \vartheta),$ *val*$(G, \sigma),$ *val*$(G, \pi)])))$ **for** u
by *force*
moreover have
val$(G,\sigma) \in M[G]$ **and** $\vartheta \in M \Longrightarrow$ *val*$(G,\vartheta) \in M[G]$ **for** ϑ
using *GenExt-def* $\langle \sigma \in M \rangle$ **by** *auto*
ultimately have
$u \in$ *domain*$(\pi) \times P \Longrightarrow u \in M \Longrightarrow$
sats$(M,$*?*$\psi,[u, P,$ *leq*, *one*$,\sigma, \pi]) \Longrightarrow$
$(\exists \vartheta \in M.\ \exists p \in P.\ u = \langle \vartheta,p \rangle \wedge$
$(p \in G \longrightarrow$
val$(G,\vartheta) \in$ *val*$(G,\pi) \wedge$ *sats*$(M[G], \varphi,$ $[$*val*$(G, \vartheta),$ *val*$(G, \sigma),$ *val*$(G, \pi)])))$ **for** u
using $\langle \pi \in M \rangle$ **by** *auto*
with $\langle$*domain*$(\pi) \times P \in M \rangle$ **have**
$\forall u \in$ *domain*$(\pi) \times P$. *sats*$(M,$*?*$\psi,[u]$ @ *?Pl1* @ $[\sigma,\pi])$ $\longrightarrow$
$(\exists \vartheta \in M.\ \exists p \in P.\ u = \langle \vartheta,p \rangle \wedge$
$(p \in G \longrightarrow$ *val*$(G, \vartheta) \in$ *val*$(G, \pi) \wedge$ *sats*$(M[G], \varphi,$ $[$*val*$(G, \vartheta),$ *val*$(G, \sigma),$ *val*$(G, \pi)])))$
by (*simp add*:*transitivity*)
then have
$\{u \in$ *domain*$(\pi) \times P$. *sats*$(M,$*?*$\psi,[u]$ @ *?Pl1* @ $[\sigma,\pi])$ $\} \subseteq$
$\{u \in$ *domain*$(\pi) \times P$. $\exists \vartheta \in M.\ \exists p \in P.\ u = \langle \vartheta,p \rangle \wedge$
$(p \in G \longrightarrow$ *val*$(G, \vartheta) \in$ *val*$(G, \pi) \wedge$ *sats*$(M[G], \varphi,$ $[$*val*$(G, \vartheta),$ *val*$(G, \sigma),$ *val*$(G, \pi)]))\}$
(**is** *?n*$\subseteq$*?m*)
by *auto*
with *val-mono* **have**
first-incl: *val*$(G,$*?n*$) \subseteq$ *val*$(G,$*?m*$)$
by *simp*
note
$\langle$*val*$(G,\pi) = c\rangle$ $\langle$*val*$(G,\sigma) = w\rangle$
with $\langle$*?*$\psi \in$*formula*$\rangle$ $\langle$*arity*$($*?*$\psi) \leq 6\rangle$ *in-M* **have**
?n$\in M$
using *six-sep-aux* **by** *simp*
from *generic* **have**
filter(G) $G \subseteq P$
unfolding *M-generic-def filter-def* **by** *simp-all*
from $\langle$*val*$(G,\pi) = c\rangle$ $\langle$*val*$(G,\sigma) = w\rangle$ **have**
val$(G,$*?m*$) =$
$\{$*val*(G,t) .. $t \in$*domain*(π) , $\exists q \in P$.
$(\exists \vartheta \in M.\ \exists p \in P.\ <t,q> = \langle \vartheta, p \rangle \wedge$

$(p \in G \longrightarrow val(G, \vartheta) \in c \wedge sats(M[G], \varphi, [val(G, \vartheta), w, c])) \wedge q \in G)\}$
using *val-of-name* **by** *auto*
also have
$... = \{val(G,t) .. t \in domain(\pi) , \exists q \in P.$
$val(G, t) \in c \wedge sats(M[G], \varphi, [val(G, t), w, c]) \wedge q \in G\}$
proof −
have
$t \in M \Longrightarrow$
$(\exists q \in P. (\exists \vartheta \in M. \exists p \in P. <t,q> = \langle \vartheta, p \rangle \wedge$
$(p \in G \longrightarrow val(G, \vartheta) \in c \wedge sats(M[G], \varphi, [val(G, \vartheta), w, c])) \wedge q \in$
$G))$
$\longleftrightarrow$
$(\exists q \in P. val(G, t) \in c \wedge sats(M[G], \varphi, [val(G, t), w, c]) \wedge q \in G)$ **for** t
by *auto*
then show *?thesis* **using** ‹$domain(\pi) \in M$› **by** (*auto simp add:transitivity*)
qed
also have
$... = \{x .. x \in c , \exists q \in P. x \in c \wedge sats(M[G], \varphi, [x, w, c]) \wedge q \in G\}$
proof
show
$... \subseteq \{x .. x \in c , \exists q \in P. x \in c \wedge sats(M[G], \varphi, [x, w, c]) \wedge q \in G\}$
by *auto*
next

{
fix x
assume
$x \in \{x .. x \in c , \exists q \in P. x \in c \wedge sats(M[G], \varphi, [x, w, c]) \wedge q \in G\}$
then have
$\exists q \in P. x \in c \wedge sats(M[G], \varphi, [x, w, c]) \wedge q \in G$
by *simp*
with ‹$val(G,\pi) = c$› **have**
$\exists q \in P. \exists t \in domain(\pi). val(G,t) = x \wedge sats(M[G], \varphi, [val(G,t), w, c]) \wedge q$
$\in G$
using *Sep-and-Replace elem-of-val* **by** *auto*
}
then show
$\{x .. x \in c , \exists q \in P. x \in c \wedge sats(M[G], \varphi, [x, w, c]) \wedge q \in G\} \subseteq ...$
using *SepReplace-iff* **by** *force*
qed
also have
$... = \{x \in c. sats(M[G], \varphi, [x, w, c])\}$
using ‹$G \subseteq P$› *G-nonempty* **by** *force*
finally have
val-m: $val(G,?m) = \{x \in c. sats(M[G], \varphi, [x, w, c])\}$ **by** *simp*
have
$val(G,?m) \subseteq val(G,?n)$
proof
fix x

```
assume
  x ∈ val(G,?m)
with val-m have
  Eq4: x ∈ {x∈c. sats(M[G], φ, [x, w, c])} by simp
with ‹val(G,π) = c› have
  x ∈ val(G,π) by simp
then have
  ∃ϑ. ∃q∈G. ⟨ϑ,q⟩∈π ∧ val(G,ϑ) =x
  using elem-of-val-pair by auto
then obtain ϑ q where
  ⟨ϑ,q⟩∈π q∈G val(G,ϑ)=x by auto
from ‹⟨ϑ,q⟩∈π› ‹π∈M› trans-M have
  ϑ∈M
  unfolding Pair-def Transset-def by auto
with ‹π∈M› ‹σ∈M› have
  [val(G,ϑ), val(G,σ), val(G,π)]∈list(M[G])
  using GenExt-def by auto
with  Eq4 ‹val(G,ϑ)=x› ‹val(G,π) = c› ‹val(G,σ) = w› ‹x ∈ val(G,π)› have
  Eq5: sats(M[G], And(Member(0,2),φ), [val(G,ϑ), val(G,σ), val(G,π)])
  by auto

with ‹ϑ∈M› ‹π∈M›  ‹σ∈M› Eq5 ‹M-generic(G)› ‹φ∈formula› have
  (∃r∈G. sats(M,forces(?χ), [P,leq,one,r,ϑ,σ,π]))
  using truth-lemma  by auto
then obtain r where
  r∈G sats(M,forces(?χ), [P,leq,one,r,ϑ,σ,π]) by auto
with ‹filter(G)› and ‹q∈G› obtain p where
  p∈G <p,q>∈leq <p,r>∈leq
  unfolding filter-def compat-in-def by force
with ‹r∈G›  ‹q∈G› ‹G⊆P› have
  p∈P r∈P q∈P p∈M
  using  P-in-M  by (auto simp add:transitivity)
with ‹φ∈formula› ‹ϑ∈M› ‹π∈M› ‹σ∈M›  ‹<p,r>∈leq›
  ‹sats(M,forces(?χ), [P,leq,one,r,ϑ,σ,π])› have
  sats(M,forces(?χ), [P,leq,one,p,ϑ,σ,π])
  using strengthening by simp
with ‹p∈P› ‹φ∈formula› ‹ϑ∈M› ‹π∈M› ‹σ∈M› have
  ∀F. M-generic(F) ∧ p ∈ F ⟶
           sats(M[F], ?χ, [val(F, ϑ), val(F, σ), val(F,π)])
  using definition-of-forces by simp
with ‹p∈P› ‹ϑ∈M›  have
  Eq6: ∃ϑ'∈M. ∃p'∈P.  ⟨ϑ,p⟩ = <ϑ',p'> ∧ (∀F. M-generic(F) ∧ p' ∈ F ⟶
           sats(M[F], ?χ, [val(F, ϑ'), val(F, σ), val(F,π)])) by auto
from ‹π∈M› ‹⟨ϑ,q⟩∈π› have
  ⟨ϑ,q⟩ ∈ M by (simp add:transitivity)
from ‹⟨ϑ,q⟩∈π› ‹ϑ∈M› ‹p∈P›  ‹p∈M› have
  ⟨ϑ,p⟩∈M ⟨ϑ,p⟩∈domain(π)×P
  using pairM by auto
with ‹ϑ∈M› Eq6 ‹p∈P› have
```

```
      sats(M,?ψ,[⟨ϑ,p⟩] @ ?Pl1 @ [σ,π])
      using Equivalence by auto
    with ‹⟨ϑ,p⟩∈domain(π)×P› have
      ⟨ϑ,p⟩∈?n by simp
    with ‹p∈G› ‹p∈P› have
      val(G,ϑ)∈val(G,?n)
      using val-of-elem[of ϑ p] by simp
    with ‹val(G,ϑ)=x› show
      x∈val(G,?n) by simp
  qed
  with val-m first-incl have
    val(G,?n) = {x∈c. sats(M[G], φ, [x, w, c])} by auto
  also have
    ... = {x∈c. sats(M[G], φ, [x, w])}
  proof −
    {
      fix x
      assume
        x∈c
      moreover from assms have
        c∈M[G] w∈M[G]
        unfolding GenExt-def by auto
      moreover with ‹x∈c› have
        x∈M[G]
        by (simp add:Transset-MG Transset-intf)
      ultimately have
        sats(M[G], φ, [x,w]@[c]) ⟷ sats(M[G], φ, [x,w])
        using phi by (rule-tac arity-sats-iff, simp-all)
    }
    then show ?thesis by auto
  qed
  finally show
    {x∈c. sats(M[G], φ, [x, w])}∈ M[G]
    using ‹?n∈M› GenExt-def by force
qed

theorem separation-in-MG:
  assumes
    φ∈formula and arity(φ) = 1 ∨ arity(φ)=2
  shows
    (∀ a∈(M[G]). separation(##M[G],λx. sats(M[G],φ,[x,a])))
proof −
  {
    fix c w
    assume
      c∈M[G] w∈M[G]
    then obtain π σ where
      val(G, π) = c val(G, σ) = w π ∈ M σ ∈ M
      using GenExt-def by auto
```

```
    with assms have
      Eq1: {x∈c. sats(M[G], φ, [x,w])} ∈ M[G]
      using Collect-sats-in-MG by auto
  }
  then show ?thesis using separation-iff rev-bexI
    unfolding is-Collect-def by force
  qed
end
end
theory Pairing-Axiom imports Names Interface begin

context forcing-data
begin

lemma valsigma :
  one ∈ G ⟹ {⟨τ,one⟩,⟨ϱ,one⟩} ∈ M ⟹
   val(G,{⟨τ,one⟩,⟨ϱ,one⟩}) = {val(G,τ),val(G,ϱ)}
  by (insert one-in-P, rule trans, subst def-val,auto simp add: Sep-and-Replace)

lemma pairing-in-MG :
  assumes M-generic(G)
  shows upair-ax(##M[G])
proof −
  {
    fix x y
    have one∈G using assms one-in-G by simp
    from assms have G⊆P
      unfolding M-generic-def and filter-def by simp
    with ‹one∈G›have one∈P using subsetD by simp
    then have one∈M using Transset-intf[OF trans-M - P-in-M] by simp
    assume x ∈ M[G] y ∈ M[G]
    then obtain τ ϱ where
      0 : val(G,τ) = x val(G,ϱ) = y ϱ ∈ M  τ ∈ M
      using GenExtD by blast
    with ‹one∈M› have ⟨τ,one⟩ ∈ M ⟨ϱ,one⟩∈M
      using pair-in-M-iff by auto
    then have 1: {⟨τ,one⟩,⟨ϱ,one⟩} ∈ M (is ?σ ∈ -)
      using upair-in-M-iff by simp
    then have val(G,?σ) ∈ M[G]
      using GenExtI by simp
    with 1 have {val(G,τ),val(G,ϱ)} ∈ M[G]
      using valsigma assms one-in-G by simp
    with 0 have {x,y} ∈ M[G] by simp
    }
    then show ?thesis unfolding upair-ax-def upair-def by auto
qed

end
end
```

```
theory Union-Axiom
  imports Names Nat-Miscellanea
begin

context forcing-data
begin

definition Union-name-body :: [i,i,i,i] ⇒ o where
  Union-name-body(P′,leq′,τ,ϑp) == (∃ σ[##M].
          ∃ q[##M]. (q∈ P′ ∧ (<σ,q> ∈ τ ∧
               (∃ r[##M].r∈P′ ∧ (<fst(ϑp),r> ∈ σ ∧ <snd(ϑp),r> ∈ leq′ ∧
<snd(ϑp),q> ∈ leq′)))))

definition Union-name-fm :: i where
  Union-name-fm ==
    Exists(∗ ϑ ∗)(
    Exists(∗ p ∗)(And(pair-fm(1,0,2),
    Exists (∗ σ∗)(
    Exists (∗ q∗)(And(Member(0,7),
      Exists (∗ σ,q ∗)(And(And(pair-fm(2,1,0),Member(0,6)),
        Exists (∗ r ∗)(And(Member(0,9),
         Exists (∗ ϑ,r ∗)(And(And(pair-fm(6,1,0),Member(0,4)),
          Exists (∗ p,r ∗)(And(And(pair-fm(6,2,0),Member(0,10)),
          Exists (∗ p,q ∗)(And(pair-fm(7,5,0),Member(0,11)))))))))))))))))

lemma Union-name-fm-type [TC]:
  Union-name-fm ∈formula
  unfolding Union-name-fm-def by simp


lemma Union-name-fm-arity :
  arity(Union-name-fm) = 4
  unfolding Union-name-fm-def upair-fm-def pair-fm-def
  by(auto simp add: nat-simp-union)

lemma sats-Union-name-fm :
  ⟦ a ∈ M; b ∈ M ; P′ ∈ M ; p ∈ M ; ϑ ∈ M ; τ ∈ M ; leq′ ∈ M ⟧ ⟹
     sats(M,Union-name-fm,[<ϑ,p>,τ,leq′,P′]@[a,b]) ⟷
     Union-name-body(P′,leq′,τ,<ϑ,p>)
  unfolding Union-name-fm-def Union-name-body-def pairM
  by (subgoal-tac <ϑ,p> ∈ M, auto simp add : pairM)


lemma domD :
  assumes τ ∈ M σ ∈ domain(τ)
  shows σ ∈ M
  using assms Transset-M trans-M
  by (simp del:setclass-iff  add:setclass-iff [symmetric])
```

definition *Union-name* :: *i* ⇒ *i* **where**
Union-name(τ) ==
{*u* ∈ *domain*(⋃(*domain*(τ))) × *P* . *Union-name-body*(*P*,*leq*,τ,*u*)}

lemma *Union-name-M* : **assumes** τ ∈ *M*
shows {*u* ∈ *domain*(⋃(*domain*(τ))) × *P* . *Union-name-body*(*P*,*leq*,τ,*u*)} ∈ *M*
unfolding *Union-name-def*
proof −
let *?P*=λ *x* . *sats*(*M*,*Union-name-fm*,[*x*,τ,*leq*]@[*P*,τ,*leq*])
let *?Q*=λ *x* . *Union-name-body*(*P*,*leq*,τ,*x*)
from ‹τ∈*M*› **have** *domain*(⋃(*domain*(τ)))∈*M* (**is** *?d* ∈ -) **using** *domain-closed* *Union-closed* **by** *simp*
then have *?d* × *P* ∈ *M* **using** *cartprod-closed P-in-M* **by** *simp*
have *arity*(*Union-name-fm*)≤*6* **using** *Union-name-fm-arity* **by** *simp*
from *assms P-in-M leq-in-M Union-name-fm-arity* **have**
[τ,*leq*] ∈ *list*(*M*) [*P*,τ,*leq*] ∈ *list*(*M*) **by** *auto*
with *assms assms P-in-M leq-in-M* ‹*arity*(*Union-name-fm*)≤*6*› **have**
∀ *u*∈*M* . *separation*(##*M*,*?P*)
using *sixp-sep*[*of Union-name-fm* τ *leq P* τ *leq*] **by** *simp*
with ‹*?d* × *P* ∈ *M*› **have** *A*:{ *u* ∈ *?d* × *P* . *?P*(*u*) } ∈ *M*
using *separation-iff* **by** *force*
{**fix** *x*
assume *x* ∈ *?d*×*P*
then have *x* = <*fst*(*x*),*snd*(*x*)> **using** *Pair-fst-snd-eq* **by** *simp*
with ‹*x*∈*?d*×*P*› ‹*?d*∈*M*› **have**
fst(*x*) ∈ *M* *snd*(*x*) ∈ *M* **using** *transM fst-type snd-type P-in-M* **by** *auto*
then have *?P*(<*fst*(*x*),*snd*(*x*)>) ⟷ *?Q*(<*fst*(*x*),*snd*(*x*)>)
using *P-in-M sats-Union-name-fm P-in-M* ‹τ∈*M*› *leq-in-M* **by** *simp*
with ‹*x* = <*fst*(*x*),*snd*(*x*)>› **have** *?P*(*x*) ⟷ *?Q*(*x*) **by** *simp*
}
then have *?P*(*x*)⟷ *?Q*(*x*) **if** *x*∈ *?d*×*P* **for** *x* **using** *that* **by** *simp*
then show *?thesis* **using** *Collect-cong A* **by** *simp*
qed

lemma *Union-abs-trans* :
assumes *Transset*(*Q*) *a* ∈ *Q* *z* ∈ *Q* ⋃ *a* = *z*
shows *big-union*(##*Q*,*a*,*z*)
proof −
{
fix *x*
assume *x* ∈ *z*
with ‹⋃ *a*=*z*› ‹*Transset*(*Q*)› ‹*a*∈*Q*› **obtain** *y* **where**
y ∈ *a* *x* ∈ *y* *y* ∈ *Q*
unfolding *Transset-def* **using** *subsetD* **by** *blast*
then have ∃ *y*[##*Q*].*x*∈*y* ∧ *y* ∈*a* **by** *auto*
}
then have *1*: *x* ∈ *z* ⟹ ∃ *y*[##*Q*].*x*∈*y* ∧ *y* ∈*a* **for** *x* **.**

with ⟨$\bigcup$ *a=z*⟩ **have** $\exists y[\#\#Q].\ y \in a \wedge x \in y \Longrightarrow x{\in}z$ **for** *x* **by** *blast*
then show *?thesis* **using** *1* **unfolding** *big-union-def* **by** *blast*
qed

lemma *Union-MG-Eq* :
assumes $a \in M[G]$ **and** $a = val(G,\tau)$ **and** *filter*(G) **and** $\tau \in M$
shows $\bigcup\ a = val(G,Union\text{-}name(\tau))$
proof −
{
fix *x*
assume $x \in \bigcup\ (val(G,\tau))$
then obtain *i* **where** $i \in val(G,\tau)\ x \in i$ **by** *blast*
with ⟨$\tau \in M$⟩ **obtain** σ *q* **where**
$q \in G <\sigma,q> \in \tau\ val(G,\sigma) = i\ \sigma \in M$
using *elem-of-val-pair domD* **by** *blast*
with ⟨$x \in i$⟩ **obtain** ϑ *r* **where**
$r \in G <\vartheta,r> \in \sigma\ val(G,\vartheta) = x\ \vartheta \in M$
using *elem-of-val-pair domD* **by** *blast*
with ⟨$<\sigma,q>\in\tau$⟩ **have** $\vartheta \in domain(\bigcup(domain(\tau)))$ **by** *auto*
with ⟨*filter*(G)⟩ ⟨$q{\in}G$⟩ ⟨$r{\in}G$⟩ **obtain** *p* **where**
A: $p \in G <p,r> \in leq <p,q> \in leq\ p \in P\ r \in P\ q \in P$
using *low-bound-filter filterD* **by** *blast*
then have $p \in M\ q{\in}M\ r{\in}M$ **using** *transM P-in-M* **by** *auto*
with *A* ⟨$<\vartheta,r> \in \sigma$⟩ ⟨$<\sigma,q> \in \tau$⟩ ⟨$\vartheta \in M$⟩ ⟨$\vartheta \in domain(\bigcup(domain(\tau)))$⟩
⟨$\sigma{\in}M$⟩ **have**
$<\vartheta,p> \in Union\text{-}name(\tau)$ **unfolding** *Union-name-def Union-name-body-def*
by *auto*
with ⟨$p{\in}P$⟩ ⟨$p{\in}G$⟩ **have** $val(G,\vartheta) \in val(G,Union\text{-}name(\tau))$
using *val-of-elem* **by** *simp*
with ⟨$val(G,\vartheta)=x$⟩ **have** $x \in val(G,Union\text{-}name(\tau))$ **by** *simp*
}
with ⟨$a=val(G,\tau)$⟩ **have** *1*: $x \in \bigcup\ a \Longrightarrow x \in val(G,Union\text{-}name(\tau))$ **for** *x* **by**
simp
{
fix *x*
assume $x \in (val(G,Union\text{-}name(\tau)))$
then obtain ϑ *p* **where**
$p \in G <\vartheta,p> \in Union\text{-}name(\tau)\ val(G,\vartheta) = x$
using *elem-of-val-pair* **by** *blast*
with ⟨*filter*(G)⟩ **have** $p{\in}P$ **using** *filterD* **by** *simp*
from ⟨$<\vartheta,p> \in Union\text{-}name(\tau)$⟩ **obtain** σ *q r* **where**
$\sigma \in domain(\tau)\ \ <\sigma,q> \in \tau\ \ <\vartheta,r> \in \sigma\ r{\in}P\ q{\in}P <p,r> \in leq <p,q> \in leq$
unfolding *Union-name-def Union-name-body-def* **by** *force*
with ⟨$p{\in}G$⟩ ⟨*filter*(G)⟩ **have** $r \in G\ q \in G$
using *filter-leqD* **by** *auto*
with ⟨$<\vartheta,r> \in \sigma$⟩ ⟨$<\sigma,q>\in\tau$⟩ ⟨$q{\in}P$⟩ ⟨$r{\in}P$⟩ **have**
$val(G,\sigma) \in val(G,\tau)\ val(G,\vartheta) \in val(G,\sigma)$
using *val-of-elem* **by** *simp+*
then have $val(G,\vartheta) \in \bigcup\ val(G,\tau)$ **by** *blast*

```
  with ⟨val(G,ϑ)=x⟩ ⟨a=val(G,τ)⟩ have
    x ∈ ⋃ a by simp
}
 with ⟨a=val(G,τ)⟩ have x ∈ val(G,Union-name(τ)) ⟹ x ∈ ⋃ a for x by blast
  then show ?thesis using 1 by blast
qed

lemma union-in-MG : assumes filter(G)
  shows Union-ax(##M[G])
  proof −
    { fix a
      assume a ∈ M[G]
      then obtain τ where τ ∈ M a=val(G,τ)     using GenExtD by blast
     then have Union-name(τ) ∈ M (is ?π ∈ -) using Union-name-M unfolding
Union-name-def by simp
      then have val(G,?π) ∈ M[G] (is ?U ∈ -) using GenExtI by simp
      with ⟨a∈M[G]⟩ ⟨τ ∈ M⟩ ⟨filter(G)⟩ ⟨?U ∈ M[G]⟩ ⟨a=val(G,τ)⟩
      have big-union(##M[G],a,?U)
        using Union-MG-Eq Union-abs-trans Transset-MG by blast
      with ⟨?U ∈ M[G]⟩ have ∃z[##M[G]]. big-union(##M[G],a,z) by force
    }
    then have Union-ax(##M[G]) unfolding Union-ax-def by force
    then show ?thesis by simp
  qed

theorem Union-MG : M-generic(G) ⟹ Union-ax(##M[G])
  by (simp add:M-generic-def union-in-MG)

end
end
theory Powerset-Axiom
  imports Separation-Axiom Pairing-Axiom Union-Axiom
begin

definition
  perm-pow :: i where
  perm-pow == {<0,3>,<1,4>,<2,5>,<3,1>,<4,0>,<5,6>}

lemma perm-pow-ftc : perm-pow ∈ 6 −||> 7 domain(perm-pow) = 6
  unfolding perm-pow-def
  by (blast intro: consI emptyI,auto)

lemma perm-pow-tc : perm-pow ∈ 6 → 7
  using FiniteFun-is-fun perm-pow-ftc
  by force

lemma perm-pow-env :
   {p,l,o,ss,fs,χ} ⊆ A ⟹ j<6 ⟹
  nth(j,[p,l,o,ss,fs,χ]) = nth(perm-pow‘j,[fs,ss,sp,p,l,o,χ])
```

apply(*subgoal-tac* $j \in nat$)
apply(*rule natE*,*simp*,*subst apply-fun*,*rule perm-pow-tc*,*simp add*:*perm-pow-def*,*simp-all*)+
apply(*subst apply-fun*,*rule perm-pow-tc*,*simp add*:*perm-pow-def*,*simp-all*,*drule ltD*,*auto*)
done

lemma (**in** *M-trivial*) *powerset-subset-Pow*:
assumes
$powerset(M,x,y)$ $\bigwedge z.\ z\in y \Longrightarrow M(z)$
shows
$y \subseteq Pow(x)$
using *assms* **unfolding** *powerset-def*
by (*auto*)

lemma (**in** *M-trivial*) *powerset-abs*:
assumes
$M(x)$ $\bigwedge z.\ z\in y \Longrightarrow M(z)$
shows
$powerset(M,x,y) \longleftrightarrow y = \{a\in Pow(x) \ .\ M(a)\}$
proof (*intro iffI equalityI*)

assume
$powerset(M,x,y)$
with *assms* **have**
$y \subseteq Pow(x)$
using *powerset-subset-Pow* **by** *simp*
with *assms* **show**
$y \subseteq \{a\in Pow(x) \ .\ M(a)\}$
by *blast*
{
fix *a*
assume
$a \subseteq x$ $M(a)$
then have
$subset(M,a,x)$ **by** *simp*
with ‹$M(a)$› ‹$powerset(M,x,y)$› **have**
$a \in y$
unfolding *powerset-def* **by** *simp*
}
then show
$\{a\in Pow(x) \ .\ M(a)\} \subseteq y$
by *auto*
next
assume
$y = \{a \in Pow(x) \ .\ M(a)\}$
then show
$powerset(M, x, y)$
unfolding *powerset-def*

```
    by simp
qed

lemma Collect-inter-Transset:
  assumes
    Transset(M) b ∈ M
  shows
    {x∈b . P(x)} = {x∈b . P(x)} ∩ M
    using assms unfolding Transset-def
  by (auto)

context G-generic begin

lemma name-components-in-M:
  assumes <σ,p>∈ϑ ϑ ∈ M
  shows   σ∈M p∈M
proof −
  from assms obtain a where
    σ ∈ a p ∈ a a∈<σ,p>
    unfolding Pair-def by auto
  moreover from assms have
    <σ,p>∈M
    using trans-M  Transset-intf[of - <σ,p>] by simp
  moreover from calculation have
    a∈M
    using trans-M  Transset-intf[of - - <σ,p>] by simp
  ultimately show
    σ∈M p∈M
    using trans-M  Transset-intf[of - - a] by simp-all
qed

lemma sats-fst-snd-in-M:
  assumes
    A∈M B∈M φ ∈ formula p∈M l∈M o∈M χ∈M
    arity(φ) ≤ 6
  shows
    {sq ∈A×B . sats(M,φ,[p,l,o,snd(sq),fst(sq),χ])} ∈ M
    (is ?ϑ ∈ M)
proof −
  have 6∈nat 7∈nat by simp-all
  let ?φ' = ren(φ)`6`7`perm-pow
  from ‹A∈M› ‹B∈M› have
    A×B ∈ M
    using cartprod-closed by simp
  from  ‹arity(φ) ≤ 6› ‹φ∈ formula› have
    ?φ' ∈ formula arity(?φ')≤7
    using perm-pow-tc ren-arity ren-tc by simp-all
  with ‹?φ' ∈ formula› have
    1: arity(Exists(Exists(And(pair-fm(0,1,2),?φ'))))≤5     (is arity(?ψ)≤5)
```

unfolding *pair-fm-def upair-fm-def*
using *nat-simp-union pred-le arity-type* **by** *auto*
{
fix *sp*
note ‹*A*×*B* ∈ *M*›
moreover assume
sp ∈ *A*×*B*
moreover from *calculation* **have**
fst(*sp*) ∈ *A* *snd*(*sp*) ∈ *B*
using *fst-type snd-type* **by** *simp-all*
ultimately have
sp ∈ *M* *fst*(*sp*) ∈ *M* *snd*(*sp*) ∈ *M*
using ‹*A*∈*M*› ‹*B*∈*M*›
by (*simp-all add*: *trans-M Transset-intf*)
note
inM = ‹*A*∈*M*› ‹*B*∈*M*› ‹*p*∈*M*› ‹*l*∈*M*› ‹*o*∈*M*› ‹*χ*∈*M*›
‹*sp*∈*M*› ‹*fst*(*sp*)∈*M*› ‹*snd*(*sp*)∈*M*›
with *1* ‹*sp* ∈ *M*› ‹*?φ′* ∈ *formula*› **have**
sats(*M*,*?ψ*,[*sp*,*p*,*l*,*o*,*χ*]@[*p*]) ⟷ *sats*(*M*,*?ψ*,[*sp*,*p*,*l*,*o*,*χ*]) (**is** *sats*(*M*,-,*?env0*@-) ⟷ -)
using *arity-sats-iff*[*of ?ψ* [*p*] *M ?env0*] **by** *auto*
also from *inM* ‹*sp* ∈ *A*×*B*› **have**
... ⟷ *sats*(*M*,*?φ′*,[*fst*(*sp*),*snd*(*sp*),*sp*,*p*,*l*,*o*,*χ*])
by *auto*
also from *inM* ‹*φ* ∈ *formula*› ‹*arity*(*φ*) ≤ *6*› **have**
... ⟷ *sats*(*M*,*φ*,[*p*,*l*,*o*,*snd*(*sp*),*fst*(*sp*),*χ*])
(**is** *sats*(-,-,*?env1*) ⟷ *sats*(-,-,*?env2*))
using *sats-iff-sats-ren*[*of φ 6 7 ?env2 M ?env1 perm-pow*] *perm-pow-tc perm-pow-env* [*of - - - - - - M*]
by *simp*
finally have
sats(*M*,*?ψ*,[*sp*,*p*,*l*,*o*,*χ*,*p*]) ⟷
sats(*M*,*φ*,[*p*,*l*,*o*,*snd*(*sp*),*fst*(*sp*),*χ*])
by *simp*
}
then have
?ϑ = {*sp*∈*A*×*B* . *sats*(*M*,*?ψ*,[*sp*,*p*,*l*,*o*,*χ*,*p*])}
by *auto*
also from *assms* ‹*A*×*B*∈*M*› **have**
... ∈ *M*
proof −
from *1* **have**
arity(*?ψ*) ≤ *6*
using *leI* **by** *simp*
moreover from ‹*?φ′* ∈ *formula*› **have**
?ψ ∈ *formula*
by *simp*
moreover note *assms* ‹*A*×*B*∈*M*›
ultimately show

$\{x \in A\times B\ .\ sats(M,\ ?\psi,\ [x,\ p,\ l,\ o,\ \chi,\ p])\} \in M$
using *sixp-sep Collect-abs separation-iff*
by *simp*
qed
finally show *?thesis* **.**
qed

lemma *Pow-inter-MG*:
assumes
$a{\in}M[G]$
shows
$Pow(a) \cap M[G] \in M[G]$
proof −
from *assms* **obtain** τ **where**
$\tau \in M\ val(G,\ \tau) = a$
using *GenExtD* **by** *blast*
let
$?Q{=}Pow(domain(\tau){\times}P) \cap M$
from ‹$\tau{\in}M$› **have**
$domain(\tau){\times}P \in M\ domain(\tau) \in M$
using *domain-closed cartprod-closed P-in-M*
by *simp-all*
then have
$?Q \in M$
proof −
from *power-ax* ‹$domain(\tau){\times}P \in M$› **obtain** Q **where**
$powerset(\#\#M,domain(\tau){\times}P,Q)\ \ Q \in M$
unfolding *power-ax-def* **by** *auto*
moreover from *calculation* **have**
$z{\in}Q\ \Longrightarrow z{\in}M$ **for** z
using *Transset-intf trans-M* **by** *blast*
ultimately have
$Q = \{a{\in}Pow(domain(\tau){\times}P)\ .\ a{\in}M\}$
using ‹$domain(\tau){\times}P \in M$› *powerset-abs*[*of* $domain(\tau){\times}P\ Q$]
by (*simp del*:*setclass-iff add*:*setclass-iff* [*symmetric*])
also have
$... = ?Q$
by *auto*
finally show
$?Q \in M$
using ‹$Q{\in}M$› **by** *simp*
qed
let
$?\pi{=}?Q{\times}\{one\}$
let
$?b{=}val(G,?\pi)$
from ‹$?Q{\in}M$› **have**
$?\pi{\in}M$
using *one-in-P P-in-M Transset-intf transM*

by (*simp del*:*setclass-iff add*:*setclass-iff* [*symmetric*])
from ‹*?π∈M*› **have**
?b ∈ *M*[*G*]
using *GenExtI* **by** *simp*
have
Pow(*a*) ∩ *M*[*G*] ⊆ *?b*
proof
fix *c*
assume
c ∈ *Pow*(*a*) ∩ *M*[*G*]
then obtain *χ* **where**
c∈M[*G*] *χ* ∈ *M val*(*G*,*χ*) = *c*
using *GenExtD* **by** *blast*
let
?ϑ={*sp* ∈*domain*(*τ*)×*P* . *sats*(*M*,*forces*(*Member*(*0*,*1*)),[*P*,*leq*,*one*,*snd*(*sp*),*fst*(*sp*),*χ*])}
have
arity(*forces*(*Member*(*0*,*1*))) = *6*
using *arity-forces* **by** *auto*
with ‹*domain*(*τ*) ∈ *M*› ‹*χ* ∈ *M*› **have**
?ϑ ∈ *M*
using *P-in-M one-in-M leq-in-M sats-fst-snd-in-M*
by *simp*
then have
?ϑ ∈ *?Q*
by *auto*
then have
val(*G*,*?ϑ*) ∈ *?b*
using *one-in-G one-in-P generic val-of-elem* [*of ?ϑ one ?π G*]
by *auto*
have
val(*G*,*?ϑ*) = *c*
proof

{
fix *x*
assume
x ∈ *val*(*G*,*?ϑ*)
then obtain *σ p* **where**
1: <*σ*,*p*>∈*?ϑ* *p*∈*G* *val*(*G*,*σ*) = *x*
using *elem-of-val-pair*
by *blast*
moreover from ‹<*σ*,*p*>∈*?ϑ*› ‹*?ϑ* ∈ *M*› **have**
σ∈M
using *name-components-in-M*[*of - - ?ϑ*] **by** *auto*
moreover from *1* **have**
sats(*M*,*forces*(*Member*(*0*,*1*)),[*P*,*leq*,*one*,*p*,*σ*,*χ*]) *p∈P*
by *simp-all*
moreover note
‹*val*(*G*,*χ*) = *c*›

ultimately have
sats(M[G],Member(0,1),[x,c])
using ‹*χ ∈ M*› *generic definition-of-forces*
by *auto*
moreover have
x∈M[G]
using ‹*val(G,σ) = x*› ‹*σ∈M*› *GenExtI* **by** *blast*
ultimately have
x∈c
using ‹*c∈M[G]*› **by** *simp*
}
then show
val(G,?ϑ) ⊆ c
by *auto*
next

{
fix *x*
assume
x ∈ c
with ‹*c ∈ Pow(a) ∩ M[G]*› **have**
x ∈ a c∈M[G] x∈M[G]
by (*auto simp add*: *Transset-intf Transset-MG*)
with ‹*val(G, τ) = a*› **obtain** *σ* **where**
σ∈domain(τ) val(G,σ) = x
using *elem-of-val*
by *blast*
moreover note ‹*x∈c*› ‹*val(G,χ) = c*›
moreover from *calculation* **have**
val(G,σ) ∈ val(G,χ)
by *simp*
moreover note ‹*c∈M[G]*› ‹*x∈M[G]*›
moreover from *calculation* **have**
sats(M[G],Member(0,1),[x,c])
by *simp*
moreover have
Member(0,1)∈formula **by** *simp*
moreover have
σ∈M
proof −
from ‹*σ∈domain(τ)*› **obtain** *p* **where**
<σ,p> ∈ τ
by *auto*
with ‹*τ∈M*› **show** *?thesis*
using *name-components-in-M* **by** *blast*
qed
moreover note ‹*χ ∈ M*›
ultimately obtain *p* **where**
p∈G sats(M,forces(Member(0,1)),[P,leq,one,p,σ,χ])

```
      using generic truth-lemma[of Member(0,1) [σ,χ] G]
      by auto
    moreover from ‹p∈G› have
      p∈P
      using generic unfolding M-generic-def filter-def by blast
    ultimately have
      <σ,p>∈?ϑ
      using ‹σ∈domain(τ)› by simp
    with ‹val(G,σ) = x› ‹p∈G› have
      x∈val(G,?ϑ)
      using val-of-elem [of - - ?ϑ] by auto
  }
  then show
    c ⊆ val(G,?ϑ)
    by auto
qed
with ‹val(G,?ϑ) ∈ ?b› show
  c∈?b
  by simp
qed
then have
  Pow(a) ∩ M[G] = {x∈?b . x⊆a & x∈M[G]}
  by auto
also from ‹a∈M[G]› have
  ... = {x∈?b . sats(M[G],subset-fm(0,1),[x,a]) & x∈M[G]}
  using Transset-MG by force
also have
  ... = {x∈?b . sats(M[G],subset-fm(0,1),[x,a])} ∩ M[G]
  by auto
also from ‹?b∈M[G]› have
  ... = {x∈?b . sats(M[G],subset-fm(0,1),[x,a])}
  using Collect-inter-Transset Transset-MG
  by simp
also have
  ... ∈ M[G]
proof −
  have
    arity(subset-fm(0,1)) ≤ 2
    by (simp add: not-lt-iff-le leI nat-union-abs1)
  moreover note
    ‹?π∈M› ‹τ∈M› ‹val(G,τ) = a›
  ultimately show ?thesis
    using Collect-sats-in-MG by auto
qed
finally show ?thesis .
qed
end

sublocale G-generic ⊆ M-trivial##M[G]
```

using *generic Union-MG pairing-in-MG zero-in-MG Transset-intf Transset-MG*
unfolding *M-trivial-def* **by** *simp*

context *G-generic* **begin**
theorem *power-in-MG* :
power-ax(##(*M*[*G*]))
unfolding *power-ax-def*
proof (*intro rallI*, *simp only*:*setclass-iff rex-setclass-is-bex*)

fix *a*
assume
$a \in M[G]$
have
$\{x \in Pow(a) . x \in M[G]\} = Pow(a) \cap M[G]$
by *auto*
also from ‹$a \in M[G]$› **have**
... $\in M[G]$
using *Pow-inter-MG* **by** *simp*
finally have
$\{x \in Pow(a) . x \in M[G]\} \in M[G]$ **.**
moreover from ‹$a \in M[G]$› **have**
powerset(##*M*[*G*], *a*, $\{x \in Pow(a) . x \in M[G]\}$)
using *powerset-abs*[*of a* $\{x \in Pow(a) . x \in M[G]\}$]
by *simp*
ultimately show
$\exists x \in M[G]$. *powerset*(##*M*[*G*], *a*, *x*)
by *auto*
qed
end
end
theory *Infinity-Axiom*
imports *Pairing-Axiom Union-Axiom*
begin

locale *G-generic* = *forcing-data* +
fixes *G* :: *i*
assumes *generic* : *M-generic*(*G*)
begin

lemma *zero-in-MG* :
$0 \in M[G]$
proof −
from *zero-in-M* **and** *elem-of-val* **have**
$0 = val(G,0)$
by *auto*
also from *GenExtI* **and** *zero-in-M* **have**
... $\in M[G]$
by *simp*
finally show *?thesis* **.**

```
qed
end

sublocale G-generic ⊆ M-trivial##M[G]
  using generic Union-MG pairing-in-MG zero-in-MG Transset-intf Transset-MG
  unfolding M-trivial-def by simp

locale G-generic-extra = G-generic + M-extra-assms
  begin
lemma infinty-in-MG : infinity-ax(##M[G])
proof −
  from infinity-ax obtain I where
   Eq1: I∈M 0 ∈ I ∀ y∈M. y ∈ I ⟶ succ(y) ∈ I
   unfolding infinity-ax-def  by auto
  then have
    check(I) ∈ M
    using check-in-M by simp
  then have
    I∈ M[G]
    using valcheck generic one-in-G one-in-P GenExtI[of check(I) G] by simp
  with ‹0∈I› have 0∈M[G] using Transset-MG Transset-intf by simp
  with ‹I∈M› have y ∈ M if y ∈ I for y
    using  Transset-intf[OF trans-M - ‹I∈M›] that by simp
  with ‹I∈M[G]› have succ(y) ∈ I ∩ M[G] if  y ∈ I for y
    using that Eq1 Transset-MG Transset-intf by blast
  with Eq1 ‹I∈M[G]› ‹0∈M[G]› show ?thesis
    unfolding infinity-ax-def by auto
qed

end
end
```