The Constructible Universe and the Relative Consistency of the Axiom of Choice

Lawrence C Paulson

June 8, 2008

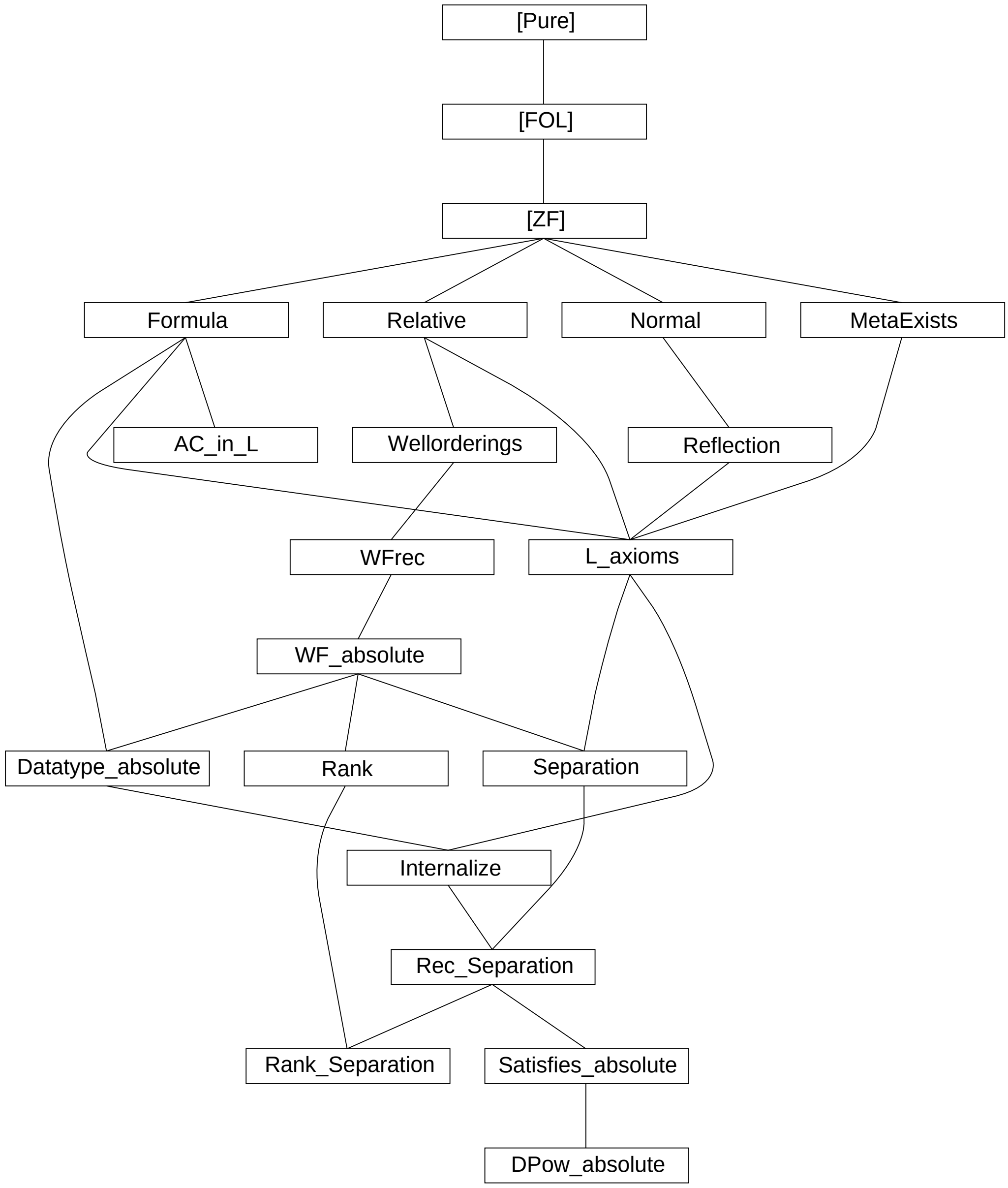
Abstract

Gödel's proof of the relative consistency of the axiom of choice [1] is one of the most important results in the foundations of mathematics. It bears on Hilbert's first problem, namely the continuum hypothesis, and indeed Gödel also proved the relative consistency of the continuum hypothesis. Just as important, Gödel's proof introduced the *inner model* method of proving relative consistency, and it introduced the concept of *constructible set*. Kunen [2] gives an excellent description of this body of work.

This Isabelle/ZF formalization demonstrates Gödel's claim that his proof can be undertaken without using metamathematical arguments, for example arguments based on the general syntactic structure of a formula. Isabelle's automation replaces the metamathematics, although it does not eliminate the requirement at least to state many tedious results that would otherwise be unnecessary.

This formalization [4] is by far the deepest result in set theory proved in any automated theorem prover. It rests on a previous formal development of the reflection theorem [3].

Contents


[Pure]
[FOL]
[ZF]
Formula
Relative
Normal
MetaExists
AC_in_L
Wellorderings
Reflection
WFrec
L_axioms
WF_absolute
Datatype_absolute
Rank
Separation
Internalize
Rec_Separation
Rank_Separation
Satisfies_absolute
DPow_absolute

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

theory *Formula* **imports** *Main* **begin**

1.1 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)) =
      (λenv ∈ list(A). bool_of_o (nth(x,env) ∈ nth(y,env)))"

  "satisfies(A,Equal(x,y)) =
      (λenv ∈ list(A). bool_of_o (nth(x,env) = nth(y,env)))"

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

  "satisfies(A,Forall(p)) =
      (λenv ∈ 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]
```

1.2 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)
```

1.2.1 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
```

1.3 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
```

1.4 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)) =
      (λnq ∈ nat. Member (incr_var(x,nq), incr_var(y,nq)))"

  "incr_bv(Equal(x,y)) =
      (λnq ∈ nat. Equal (incr_var(x,nq), incr_var(y,nq)))"

  "incr_bv(Nand(p,q)) =
      (λnq ∈ nat. Nand (incr_bv(p)`nq, incr_bv(q)`nq))"

  "incr_bv(Forall(p)) =
      (λnq ∈ 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 ∈ nat; y ∈ nat; nq ≤ x |]
      ==> succ(x) ∪ incr_var(y,nq) = succ(x ∪ 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 ∪ succ(x) = succ(x ∪ 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 ∈ formula
   ==> ∀n ∈ 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
```

1.5 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
```

1.6 Definable Powerset

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

definition

```
  DPow :: "i => i" where
  "DPow(A) == {X ∈ Pow(A).
               ∃env ∈ list(A). ∃p ∈ formula.
                 arity(p) ≤ succ(length(env)) &
                 X = {x∈A. sats(A, p, Cons(x,env))}}"

lemma DPowI:
  "[|env ∈ list(A);  p ∈ formula;  arity(p) ≤ succ(length(env))|]
   ==> {x∈A. sats(A, p, Cons(x,env))} ∈ DPow(A)"
by (simp add: DPow_def, blast)
```

With this rule we can specify p later.

```
lemma DPowI2 [rule_format]:
  "[|∀x∈A. P(x) <-> sats(A, p, Cons(x,env));
     env ∈ list(A);  p ∈ formula;  arity(p) ≤ succ(length(env))|]
   ==> {x∈A. P(x)} ∈ DPow(A)"
by (simp add: DPow_def, blast)

lemma DPowD:
  "X ∈ DPow(A)
   ==> X <= A &
       (∃env ∈ list(A).
        ∃p ∈ formula. arity(p) ≤ succ(length(env)) &
                      X = {x∈A. sats(A, p, Cons(x,env))})"
by (simp add: DPow_def)

lemmas DPow_imp_subset = DPowD [THEN conjunct1]


lemma "[| p ∈ formula; env ∈ list(A); arity(p) ≤ succ(length(env))
|]
       ==> {x∈A. sats(A, p, Cons(x,env))} ∈ DPow(A)"
by (blast intro: DPowI)

lemma DPow_subset_Pow: "DPow(A) <= Pow(A)"
by (simp add: DPow_def, blast)

lemma empty_in_DPow: "0 ∈ 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 ∈ DPow(A) ==> (A-X) ∈ 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 ∈ DPow(A); Y ∈ DPow(A) |] ==> X Int Y ∈ 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 ∈ DPow(A); Y ∈ DPow(A) |] ==> X Un Y ∈ DPow(A)"
apply (subgoal_tac "X Un Y = A - ((A-X) Int (A-Y))")
apply (simp add: Int_in_DPow Compl_in_DPow)
apply (simp add: DPow_def, blast)
done

lemma singleton_in_DPow: "a ∈ A ==> {a} ∈ 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 ∈ A; X ∈ DPow(A) |] ==> cons(a,X) ∈ DPow(A)"
apply (rule cons_eq [THEN subst])
apply (blast intro: singleton_in_DPow Un_in_DPow)
done


lemma Fin_into_DPow: "X ∈ Fin(A) ==> X ∈ 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) <= 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
```

1.7 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.

1.7.1 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
```

1.7.2 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
```

1.7.3 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
```

1.8 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) = (UN 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
```

1.8.1 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
```

1.8.2 Monotonicity

Kunen's VI 1.6 (b)

```
lemma Lset_mono [rule_format]:
     "ALL j. i<=j --> Lset(i) <= Lset(j)"
proof (induct i rule: eps_induct, intro allI impI)
```

```
  fix x j
  assume "∀y∈x. ∀j. y ⊆ j ⟶ Lset(y) ⊆ Lset(j)"
     and "x ⊆ j"
  thus "Lset(x) ⊆ 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]:
     "ALL j. i:j --> Lset(i) <= Lset(j)"
proof (induct i rule: eps_induct, intro allI impI)
  fix x j
  assume "∀y∈x. ∀j. y ∈ j ⟶ Lset(y) ⊆ Lset(j)"
     and "x ∈ j"
  thus "Lset(x) ⊆ 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 ⊆ Lset(i); i < j|] ==> A ⊆ Lset(j)"
by (blast dest: ltD [THEN Lset_mono_mem])
```

1.8.3 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)) <= Lset(succ(i))"
by (subst Lset, rule succI1 [THEN RepFunI, THEN Union_upper])

lemma Lset_succ_subset2: "Lset(succ(i)) <= 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(⋃(X)) = (⋃y∈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
```

1.8.4 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
```

1.8.5 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
```

1.9 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) ⊆ DPow(Lset(x)) ∩ 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 ⊆ Lset(i)"
by (subst Ords_of_Lset_eq [symmetric], assumption, fast)

lemma Ord_in_Lset: "Ord(i) ==> i ∈ 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)
```

1.9.1 Unions

```
lemma Union_in_Lset:
     "X ∈ Lset(i) ==> Union(X) ∈ 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 $\exists y.\ y \in X \wedge x \in y$

```
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(Union(X))"
by (simp add: L_def, blast dest: Union_in_Lset)
```

1.9.2 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], standard]
lemmas Lset_UnI2 = Un_upper2 [THEN Lset_mono [THEN subsetD], standard]
```

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 Un 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 ∈ Lset(i); Ord(i)|] ==> L(x)"
by (simp add: L_def, blast)

lemma L_D: "L(x) ==> ∃i. Ord(i) & x ∈ 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 ∈ Lset(i) --> 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 ∈ Lset(i) <-> 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="λi. x ∈ Lset(succ(i))" in LeastI, assumption)
apply (blast intro!: le_imp_subset Lset_mono [THEN subsetD])
done

lemma Lset_succ_lrank_iff [simp]: "x ∈ Lset(succ(lrank(x))) <-> 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) ∈ 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)
```

1.9.3 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)
```

1.10 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
```

2 Relativization and Absoluteness

```
theory Relative imports Main begin
```

2.1 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 ∈ z & b ∈ z & (∀x[M]. x∈z --> x = a | x = b)"

definition
  pair :: "[i=>o,i,i,i] => o" where
    "pair(M,a,b,z) == ∃x[M]. upair(M,a,a,x) &
                     (∃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) == ∀x[M]. x ∈ z <-> x ∈ a | x ∈ b"

definition
  is_cons :: "[i=>o,i,i,i] => o" where
    "is_cons(M,a,b,z) == ∃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_cons(M,a,a,z)"

definition
  number1 :: "[i=>o,i] => o" where
    "number1(M,a) == ∃x[M]. empty(M,x) & successor(M,x,a)"

definition
  number2 :: "[i=>o,i] => o" where
    "number2(M,a) == ∃x[M]. number1(M,x) & successor(M,x,a)"
```

definition
```
  number3 :: "[i=>o,i] => o" where
    "number3(M,a) == ∃x[M]. number2(M,x) & successor(M,x,a)"
```

definition
```
  powerset :: "[i=>o,i,i] => o" where
    "powerset(M,A,z) == ∀x[M]. x ∈ z <-> subset(M,x,A)"
```

definition
```
  is_Collect :: "[i=>o,i,i=>o,i] => o" where
    "is_Collect(M,A,P,z) == ∀x[M]. x ∈ z <-> x ∈ A & P(x)"
```

definition
```
  is_Replace :: "[i=>o,i,[i,i]=>o,i] => o" where
    "is_Replace(M,A,P,z) == ∀u[M]. u ∈ z <-> (∃x[M]. x∈A & P(x,u))"
```

definition
```
  inter :: "[i=>o,i,i,i] => o" where
    "inter(M,a,b,z) == ∀x[M]. x ∈ z <-> x ∈ a & x ∈ b"
```

definition
```
  setdiff :: "[i=>o,i,i,i] => o" where
    "setdiff(M,a,b,z) == ∀x[M]. x ∈ z <-> x ∈ a & x ∉ b"
```

definition
```
  big_union :: "[i=>o,i,i] => o" where
    "big_union(M,A,z) == ∀x[M]. x ∈ z <-> (∃y[M]. y∈A & x ∈ y)"
```

definition
```
  big_inter :: "[i=>o,i,i] => o" where
    "big_inter(M,A,z) ==
             (A=0 --> z=0) &
             (A≠0 --> (∀x[M]. x ∈ z <-> (∀y[M]. y∈A --> x ∈ y)))"
```

definition
```
  cartprod :: "[i=>o,i,i,i] => o" where
    "cartprod(M,A,B,z) ==
        ∀u[M]. u ∈ z <-> (∃x[M]. x∈A & (∃y[M]. y∈B & pair(M,x,y,u)))"
```

definition
```
  is_sum :: "[i=>o,i,i,i] => o" where
    "is_sum(M,A,B,Z) ==
       ∃A0[M]. ∃n1[M]. ∃s1[M]. ∃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) == ∃zero[M]. empty(M,zero) & pair(M,zero,a,z)"
```

definition
```
is_Inr :: "[i=>o,i,i] => o" where
  "is_Inr(M,a,z) == ∃n1[M]. number1(M,n1) & pair(M,n1,a,z)"
```

definition
```
is_converse :: "[i=>o,i,i] => o" where
  "is_converse(M,r,z) ==
     ∀x[M]. x ∈ z <->
          (∃w[M]. w∈r & (∃u[M]. ∃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) ==
     ∀x[M]. x ∈ z <-> (∃w[M]. w∈r & (∃y[M]. y∈A & pair(M,x,y,w)))"
```

definition
```
is_domain :: "[i=>o,i,i] => o" where
  "is_domain(M,r,z) ==
     ∀x[M]. x ∈ z <-> (∃w[M]. w∈r & (∃y[M]. pair(M,x,y,w)))"
```

definition
```
image :: "[i=>o,i,i,i] => o" where
  "image(M,r,A,z) ==
     ∀y[M]. y ∈ z <-> (∃w[M]. w∈r & (∃x[M]. x∈A & pair(M,x,y,w)))"
```

definition
```
is_range :: "[i=>o,i,i] => o" where
```
— the cleaner `∃r'[M]. is_converse(M, r, r') ∧ is_domain(M, r', z)` unfortunately needs an instance of separation in order to prove `M(converse(r))`.

```
  "is_range(M,r,z) ==
     ∀y[M]. y ∈ z <-> (∃w[M]. w∈r & (∃x[M]. pair(M,x,y,w)))"
```

definition
```
is_field :: "[i=>o,i,i] => o" where
  "is_field(M,r,z) ==
     ∃dr[M]. ∃rr[M]. is_domain(M,r,dr) & is_range(M,r,rr) &
                     union(M,dr,rr,z)"
```

definition
```
is_relation :: "[i=>o,i] => o" where
  "is_relation(M,r) ==
     (∀z[M]. z∈r --> (∃x[M]. ∃y[M]. pair(M,x,y,z)))"
```

definition
```
is_function :: "[i=>o,i] => o" where
  "is_function(M,r) ==
     ∀x[M]. ∀y[M]. ∀y'[M]. ∀p[M]. ∀p'[M].
```

```
        pair(M,x,y,p) --> pair(M,x,y',p') --> p∈r --> p'∈r --> y=y'"
```

definition

```
  fun_apply :: "[i=>o,i,i,i] => o" where
    "fun_apply(M,f,x,y) ==
        (∃xs[M]. ∃fxs[M].
         upair(M,x,x,xs) & image(M,f,xs,fxs) & big_union(M,fxs,y))"
```

definition

```
  typed_function :: "[i=>o,i,i,i] => o" where
    "typed_function(M,A,B,r) ==
        is_function(M,r) & is_relation(M,r) & is_domain(M,r,A) &
        (∀u[M]. u∈r --> (∀x[M]. ∀y[M]. pair(M,x,y,u) --> y∈B))"
```

definition

```
  is_funspace :: "[i=>o,i,i,i] => o" where
    "is_funspace(M,A,B,F) ==
        ∀f[M]. f ∈ F <-> typed_function(M,A,B,f)"
```

definition

```
  composition :: "[i=>o,i,i,i] => o" where
    "composition(M,r,s,t) ==
        ∀p[M]. p ∈ t <->
               (∃x[M]. ∃y[M]. ∃z[M]. ∃xy[M]. ∃yz[M].
                pair(M,x,z,p) & pair(M,x,y,xy) & pair(M,y,z,yz) &
                xy ∈ s & yz ∈ r)"
```

definition

```
  injection :: "[i=>o,i,i,i] => o" where
    "injection(M,A,B,f) ==
        typed_function(M,A,B,f) &
        (∀x[M]. ∀x'[M]. ∀y[M]. ∀p[M]. ∀p'[M].
          pair(M,x,y,p) --> pair(M,x',y,p') --> p∈f --> p'∈f --> x=x')"
```

definition

```
  surjection :: "[i=>o,i,i,i] => o" where
    "surjection(M,A,B,f) ==
        typed_function(M,A,B,f) &
        (∀y[M]. y∈B --> (∃x[M]. x∈A & fun_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) ==
        ∀x[M]. x ∈ z <-> (x ∈ r & (∃u[M]. u∈A & (∃v[M]. pair(M,u,v,x))))"
```

definition

```
transitive_set :: "[i=>o,i] => o" where
  "transitive_set(M,a) == ∀x[M]. x∈a --> subset(M,x,a)"
```

definition

```
ordinal :: "[i=>o,i] => o" where
   — an ordinal is a transitive set of transitive sets
  "ordinal(M,a) == transitive_set(M,a) & (∀x[M]. x∈a --> transitive_set(M,x))"
```

definition

```
limit_ordinal :: "[i=>o,i] => o" where
  — a limit ordinal is a non-empty, successor-closed ordinal
  "limit_ordinal(M,a) ==
      ordinal(M,a) & ~ empty(M,a) &
      (∀x[M]. x∈a --> (∃y[M]. y∈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_ordinal(M,a) ==
      ordinal(M,a) & ~ empty(M,a) & ~ limit_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) ==
       ∀x[M]. ∀y[M]. ∀z[M]. ∀u[M].
         x∈A --> y∈B --> z∈C --> is_f(x,y,z,u) <-> 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) ==
       ∀u[M]. ∀x[M]. ∀y[M]. ∀z[M]. ∀a[M]. is_f(u,x,y,z,a) <-> a = f(u,x,y,z)"*

Useful when absoluteness reasoning has replaced the predicates by terms

lemma *triv_Relation1:*
     "Relation1(M, A, λx y. y = f(x), f)"
by *(simp add: Relation1_def)*

lemma *triv_Relation2:*
     "Relation2(M, A, B, λx y a. a = f(x,y), f)"
by *(simp add: Relation2_def)*

2.2 The relativized ZF axioms

definition
  extensionality :: "(i=>o) => o" **where**
    *"extensionality(M) ==
        ∀x[M]. ∀y[M]. (∀z[M]. z ∈ x <-> z ∈ y) --> 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) ==
        ∀z[M]. ∃y[M]. ∀x[M]. x ∈ y <-> x ∈ z & P(x)"
```

```
definition
  upair_ax :: "(i=>o) => o" where
    "upair_ax(M) == ∀x[M]. ∀y[M]. ∃z[M]. upair(M,x,y,z)"
```

```
definition
  Union_ax :: "(i=>o) => o" where
    "Union_ax(M) == ∀x[M]. ∃z[M]. big_union(M,x,z)"
```

```
definition
  power_ax :: "(i=>o) => o" where
    "power_ax(M) == ∀x[M]. ∃z[M]. powerset(M,x,z)"
```

```
definition
  univalent :: "[i=>o, i, [i,i]=>o] => o" where
    "univalent(M,A,P) ==
        ∀x[M]. x∈A --> (∀y[M]. ∀z[M]. P(x,y) & P(x,z) --> 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))"
```

2.3 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, λx y. y = f(x))"
by (simp add: univalent_def)

lemma univalent_conjI2 [intro,simp]:
     "univalent(M,A,Q) ==> univalent(M, A, λx 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) <-> P'(x,y) |]
      ==> strong_replacement(M, %x y. P(x,y)) <->
          strong_replacement(M, %x y. P'(x,y))"
by (simp add: strong_replacement_def)
```

The extensionality axiom

```
lemma "extensionality(λx. x ∈ 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 ∈ Vfrom(A,j);  Transset(A) |] ==> Collect(X,P) ∈ Vfrom(A,
succ(j))"
apply (drule Transset_Vfrom)
apply (rule subset_mem_Vfrom)
apply (unfold Transset_def, blast)
```

```
done

lemma Collect_in_VLimit:
     "[| X ∈ Vfrom(A,i);  Limit(i);  Transset(A) |]
      ==> Collect(X,P) ∈ Vfrom(A,i)"
apply (rule Limit_VfromE, assumption+)
apply (blast intro: Limit_has_succ VfromI Collect_in_Vfrom)
done

lemma Collect_in_univ:
     "[| X ∈ univ(A);  Transset(A) |] ==> Collect(X,P) ∈ univ(A)"
by (simp add: univ_def Collect_in_VLimit Limit_nat)

lemma "separation(λx. x ∈ 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(λx. x ∈ univ(0))"
apply (simp add: upair_ax_def upair_def)
apply (blast intro: doubleton_in_univ)
done
```

Union axiom

```
lemma "Union_ax(λx. x ∈ univ(0))"
apply (simp add: Union_ax_def big_union_def, clarify)
apply (rule_tac x="⋃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 ∈ univ(A);  Transset(A) |] ==> Pow(X) ∈ univ(A)"
apply (simp add: univ_def Pow_in_VLimit Limit_nat)
done

lemma "power_ax(λx. x ∈ 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(λx. x ∈ 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(λx. x ∈ 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

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

```
lemma image_iff_Collect: "r `` A = {y ∈ Union(Union(r)). ∃p∈r. ∃x∈A.
p=<x,y>}"
apply (rule equalityI, auto)
apply (simp add: Pair_def, blast)
done
```

```
lemma vimage_iff_Collect:
     "r -`` A = {x ∈ Union(Union(r)). ∃p∈r. ∃y∈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) ==
       ∀p[M]. p ∈ r <-> (∃x[M]. x∈A & (∃y[M]. y∈A & x∈y & pair(M,x,y,p)))"
```

2.5 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∈x; M(x) |] ==> M(y)"
      and upair_ax:         "upair_ax(M)"
      and Union_ax:         "Union_ax(M)"
      and power_ax:         "power_ax(M)"
      and replacement:      "replacement(M,P)"
      and M_nat [iff]:      "M(nat)"
```

Automatically discovers the proof using `transM`, `nat_0I` and `M_nat`.

```
lemma (in M_trivial) nonempty [simp]: "M(0)"
by (blast intro: transM)

lemma (in M_trivial) rall_abs [simp]:
     "M(A) ==> (∀x[M]. x∈A --> P(x)) <-> (∀x∈A. P(x))"
```

by *(blast intro: transM)*

lemma *(***in** *M_trivial) rex_abs [simp]:*
    "M(A) ==> (∃x[M]. x∈A & P(x)) <-> (∃x∈A. P(x))"
by *(blast intro: transM)*

lemma *(***in** *M_trivial) ball_iff_equiv:*
    "M(A) ==> (∀x[M]. (x∈A <-> P(x))) <->
        (∀x∈A. P(x)) & (∀x. P(x) --> M(x) --> x∈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∈A <-> x∈B; M(A); M(B) |] ==> A=B"
by *(blast intro!: equalityI dest: transM)*

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

lemma *(***in** *M_trivial) empty_abs [simp]:*
    "M(z) ==> empty(M,z) <-> 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) <-> A ⊆ 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 ZF.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: ZF.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
```

2.5.2 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 Un 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) <-> z = a Int 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) <-> 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_union(M,A,z) <-> z = Union(A)"
apply (simp add: big_union_def)
apply (blast intro!: equalityI dest: transM)
done
```

```
lemma (in M_trivial) Union_closed [intro,simp]:
     "M(A) ==> M(Union(A))"
by (insert Union_ax, simp add: Union_ax_def)

lemma (in M_trivial) Un_closed [intro,simp]:
     "[| M(A); M(B) |] ==> M(A Un 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_cons(M,a,b,z) <-> 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) <-> z = succ(a)"
by (simp add: successor_def, blast)

lemma (in M_trivial) succ_in_M_iff [iff]:
     "M(succ(a)) <-> M(a)"
apply (simp add: succ_def)
apply (blast intro: transM)
done
```

2.5.3 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 intro!: equalityI dest: transM)
done
```

Probably the premise and conclusion are equivalent

```
lemma (in M_trivial) strong_replacementI [rule_format]:
    "[| ∀B[M]. separation(M, %u. ∃x[M]. x∈B & P(x,u)) |]
```

```
     ==> strong_replacement(M,P)"
apply (simp add: strong_replacement_def, clarify)
apply (frule replacementD [OF replacement], assumption, clarify)
apply (drule_tac x=A in rspec, clarify)
apply (drule_tac z=Y in separationD, assumption, clarify)
apply (rule_tac x=y in rexI, force, assumption)
done
```

2.5.4 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, λx y. y = f(x)); M(A); ∀x∈A. M(f(x)) |]
 ==> M(RepFun(A,f))"
```
apply *(simp add: RepFun_def)*
apply *(rule strong_replacement_closed)*
apply *(auto dest: transM simp add: univalent_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, λx 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

2.5.5 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, λx y. y = <x,b(x)>); M(A); ∀x∈A. M(b(x))
|]
 ==> M(λx∈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, λx 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
```

2.5.6 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)
```

2.6 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_ordinal(M,a) <-> Ord(a) & (∃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); ~ Limit(a); ∀x∈a. ~ Limit(x) |] ==> a ∈ nat"
by (induct a rule: trans_induct3, simp_all)

lemma (in M_trivial) finite_ordinal_abs [simp]:
     "M(a) ==> finite_ordinal(M,a) <-> a ∈ 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); ∀x∈a. ~ 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) <-> 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) <-> a = 1"
by (simp add: number1_def)

lemma (in M_trivial) number2_abs [simp]:
     "M(a) ==> number2(M,a) <-> a = succ(1)"
by (simp add: number2_def)

lemma (in M_trivial) number3_abs [simp]:
     "M(a) ==> number3(M,a) <-> a = succ(succ(1))"
by (simp add: number3_def)
```

Kunen continued to 20...

2.7 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))"

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 Un 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 Un B" and P="λx. rex(M,?Q(x))" in rspec)
apply (blast, clarify)
apply (drule_tac x=z and P="λx. rex(M,?Q(x))" 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 Un B" and P="λx. rex(M,?Q(x))" in rspec)
apply (blast, clarify)
apply (drule_tac x=z and P="λx. rex(M,?Q(x))" 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)) <-> M(a)"
by (simp add: Inl_def)

lemma (in M_trivial) Inl_abs [simp]:
     "M(Z) ==> is_Inl(M,a,Z) <-> (Z = Inl(a))"
by (simp add: is_Inl_def Inl_def)

lemma (in M_trivial) Inr_in_M_iff [iff]:
     "M(Inr(a)) <-> M(a)"
by (simp add: Inr_def)

lemma (in M_trivial) Inr_abs [simp]:
     "M(Z) ==> is_Inr(M,a,Z) <-> (Z = Inr(a))"
by (simp add: is_Inr_def Inr_def)
```

2.7.1 converse of a relation

```
lemma (in M_basic) M_converse_iff:
     "M(r) ==>
      converse(r) =
      {z ∈ Union(Union(r)) * Union(Union(r)).
       ∃p∈r. ∃x[M]. ∃y[M]. p = ⟨x,y⟩ & z = ⟨y,x⟩}"
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_converse(M,r,z) <-> 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
```

2.7.2 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) <-> 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)*

2.7.3 Domain, range and field

lemma *(***in** *M_basic) domain_abs [simp]:*
```
"[| M(r); M(z) |] ==> is_domain(M,r,z) <-> 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) <-> 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) <-> z = field(r)"
```
by *(simp add: domain_closed range_closed is_field_def field_def)*

lemma *(***in** *M_basic) field_closed [intro,simp]:*
```
"M(r) ==> M(field(r))"
```
by *(simp add: domain_closed range_closed Un_closed field_def)*

2.7.4 Relations, functions and application

lemma *(***in** *M_basic) relation_abs [simp]:*
```
"M(r) ==> is_relation(M,r) <-> 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_function(M,r) <-> 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_apply(M,f,x,y) <-> 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_function(M,A,B,f) <-> f ∈ 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) <-> f ∈ inj(A,B)"
apply (simp add: injection_def apply_iff inj_def apply_closed)
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) <-> f ∈ 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) <-> f ∈ bij(A,B)"
by (simp add: bijection_def bij_def)
```

2.7.5 Composition of relations

```
lemma (in M_basic) M_comp_iff:
     "[| M(r); M(s) |]
      ==> r O s =
          {xz ∈ domain(s) * range(r).
            ∃x[M]. ∃y[M]. ∃z[M]. xz = ⟨x,z⟩ & ⟨x,y⟩ ∈ s & ⟨y,z⟩ ∈ 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) <-> 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) <-> 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 ∈ r . ∃x∈A. ∃y[M]. z = ⟨x, y⟩}"
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_inter(M,A,z) <-> z = Inter(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(Inter(A))"
by (insert Inter_separation, simp add: Inter_def)

lemma (in M_basic) Int_closed [intro,simp]:
     "[| M(A); M(B) |] ==> M(A Int 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)
```

2.7.6 Some Facts About Separation Axioms

```
lemma (in M_basic) separation_conj:
     "[|separation(M,P); separation(M,Q)|] ==> separation(M, λz. 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) Un Collect(A,Q) = Collect(A, %x. P(x) | Q(x))"
by blast

lemma Diff_Collect_eq:
     "A - Collect(A,P) = Collect(A, %x. ~ P(x))"
by blast

lemma (in M_trivial) Collect_rall_eq:
     "M(Y) ==> Collect(A, %x. ∀y[M]. y∈Y --> P(x,y)) =
               (if Y=0 then A else (⋂y ∈ Y. {x ∈ A. P(x,y)}))"
apply simp
apply (blast intro!: equalityI dest: transM)
done

lemma (in M_basic) separation_disj:
     "[|separation(M,P); separation(M,Q)|] ==> separation(M, λz. P(z)
| 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, λz. ~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, λz. 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, λx. P(x,y));
        ∀z[M]. strong_replacement(M, λx y. y = {u ∈ z . P(u,x)})|]
      ==> separation(M, λx. ∀y[M]. y∈Y --> P(x,y))"
apply (simp del: separation_closed rall_abs
         add: separation_iff Collect_rall_eq)
apply (blast intro!: Inter_closed RepFun_closed dest: transM)
done
```

2.7.7 Functions and function space

The assumption $M(A \to B)$ 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
```

2.8 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)*

2.9 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))"

2.9.1 *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)
```

2.9.2 *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
```

2.9.3 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_hd(M,Cons(a,l),Z) <-> Z = a"
by (force simp add: is_hd_def)

lemma (in M_trivial) hd_abs [simp]:
     "[|M(x); M(y)|] ==> is_hd(M,x,y) <-> 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_tl(M,[],Z) <-> Z = []"
by (simp add: is_tl_def)

lemma (in M_trivial) is_tl_Cons:
     "[|M(a); M(l)|] ==> is_tl(M,Cons(a,l),Z) <-> Z = l"
by (force simp add: is_tl_def)

lemma (in M_trivial) tl_abs [simp]:
     "[|M(x); M(y)|] ==> is_tl(M,x,y) <-> 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_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 ∈ nat ==> tl'^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

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, λx. ~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, λx. 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)
```

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: apply_closed 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) <->
           f ∈ r -`` {a} → range(f) &
           (∀x ∈ 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); ⟨x,i⟩ ∈ f|] ==> ⟨x,
a⟩ ∈ r"
by (blast dest: is_recfun_type fun_is_rel)

lemma trans_Int_eq:
      "[| trans(r); <y,x> ∈ r |] ==> r -`` {x} ∩ 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'> ∈ r'; relation(g); domain(g) <= 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'> ∈ r'; relation(g); domain(g) <= r' -``{x} |]
             ==> H(x,g) = H'(x,g) |]
   ==> is_recfun(r,a,H,f) <-> 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 ∈ r -`` {a} → range(f); g ∈ r -`` {b} → range(g);
        M(r); M(f); M(g); M(a); M(b) |]
     ==> separation(M, λx. ¬ (⟨x, a⟩ ∈ r ⟶ ⟨x, b⟩ ∈ r ⟶ 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_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_recfun(r,a,H,f);  is_recfun(r,b,H,g);
       wellfounded(M,r);  trans(r);
       M(f); M(g); M(r); M(x); M(a); M(b) |]
     ==> <x,a> ∈ r --> <x,b> ∈ r --> 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 "∀y ∈ r-``{x1}. ALL z. <y,z>∈f <-> <y,z>∈g")
 apply (blast intro: transD)
apply (simp add: apply_iff)
apply (blast intro: transD sym)
done

lemma (in M_basic) is_recfun_cut:
    "[|is_recfun(r,a,H,f);  is_recfun(r,b,H,g);
       wellfounded(M,r); trans(r);
       M(f); M(g); M(r); <b,a> ∈ r |]
      ==> 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)")+  — 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, λx 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, 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))|]
      ==> ∃f[M]. is_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, λx. ~ (∃f[M]. is_recfun(r, x, H, f)));
       strong_replacement(M, λx z.
          ∃y[M]. ∃g[M]. pair(M,x,y,z) & is_recfun(r,x,H,g) & y = H(x,g));

       M(r);  M(a);
       ∀x[M]. ∀g[M]. function(g) --> M(H(x,g)) |]
      ==> ∃f[M]. is_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="lam 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="lam 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)"

lemma (in M_basic) 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)))"
by (simp add: rtran_closure_mem_def Ord_succ_mem_iff nat_0_le [THEN ltD])



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)"
```

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)
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 comp_closed rtrancl_closed)
```

lemma *(in M_trancl) trancl_abs [simp]:*
```
     "[| M(r); M(z) |] ==> tran_closure(M,r,z) <-> z = trancl(r)"
by (simp add: tran_closure_def trancl_def)
```

lemma *(in M_trancl) wellfounded_trancl_separation':*
```
     "[| M(r); M(Z) |] ==> separation (M, λx. ∃w[M]. w ∈ Z & <w,x> ∈
r^+)"
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 <= A |] ==> wf[A](r^+)"
apply (simp add: wf_on_def wf_def)
apply (safe intro!: equalityI)
```

```
apply (drule_tac x = "{x∈A. ∃w. ⟨w,x⟩ ∈ r^+ & w ∈ Z}" in spec)
apply (blast elim: tranclE)
done

lemma (in M_trancl) wellfounded_on_trancl:
     "[| wellfounded_on(M,A,r);  r-``A <= A; M(r); M(A) |]
      ==> wellfounded_on(M,A,r^+)"
apply (simp add: wellfounded_on_def)
apply (safe intro!: equalityI)
apply (rename_tac Z x)
apply (subgoal_tac "M({x∈A. ∃w[M]. w ∈ Z & ⟨w,x⟩ ∈ r^+})")
 prefer 2
 apply (blast intro: wellfounded_trancl_separation')
apply (drule_tac x = "{x∈A. ∃w[M]. w ∈ Z & ⟨w,x⟩ ∈ r^+}" 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^+)"
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, 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))|]
   ==> wfrec(r,a,H) = z <->
       (∃f[M]. is_recfun(r^+, a, λx 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^+" a "λx 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^+` 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);
     ∀x[M]. ∀g[M]. function(g) --> M(H(x,g))|]
   ==> wfrec(r,a,H) = z <-> (∃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);
     ∀x[M]. ∀g[M]. function(g) --> M(H(x,g))|]
   ==> is_wfrec(M,MH,r,a,z) <-> 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, λx 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="λx∈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, λx z.
          ∃y[M]. pair(M,x,y,z) & (∃g[M]. is_recfun(r,x,H,g) & y = H(x,g)))
<->
      strong_replacement(M,
           λx 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 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) ==> (⋃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) = (⋃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(λX. F(X) + G(X))"
by (simp add: contin_def, blast)

lemma prod_contin: "[|contin(F); contin(G)|] ==> contin(λX. 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(λX. A)"
by (simp add: contin_def directed_def)

lemma id_contin: "contin(λX. 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, λm 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, λm. F(g`m), n)"
```

by *(simp add: nat_case_abs [of _ "λm. F(g ` m)"] relation1_def iterates_MH_def)*

lemma *(in M_basic) iterates_imp_wfrec_replacement:*

```
  "[|relation1(M,isF,F); n ∈ nat; iterates_replacement(M,isF,v)|]
   ==> wfrec_replacement(M, λn f z. z = nat_case(v, λm. 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 nat_into_M 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 _ _ _ _ "λn g. nat_case(v, λm. 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 nat_into_M relation2_def iterates_MH_abs iterates_nat_def recursor_def transrec_def eclose_sing_Ord_eq nat_into_M trans_wfrec_closed [of _ _ _ "λn g. nat_case(v, λm. 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_mono(univ(A), λX. {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(λX. {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), λX. {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) = (⋃n∈nat. (λX. {0} + A*X) ^ 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), λX. ((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(λX. ((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), λX. ((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. (λX. ((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_basic) 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) == (λX. {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 ∈ list(A) ==> ∀n∈nat. length(l) < n --> l ∈ list_N(A, n)"
apply (induct_tac l)
apply (force elim: natE)+
done

lemma list_N_imp_eq_length:
      "[|n ∈ nat; l ∉ list_N(A, n); l ∈ list_N(A, succ(n))|]
       ==> 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 `λx. Vset(x)`, neither of which is absolute.

```
lemma (in M_trivial) list_rec_eq:
  "l ∈ list(A) ==>
   list_rec(a,g,l) =
   transrec (succ(length(l)),
      λx h. Lambda (list(A),
                    list_case' (a,
                           λa 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_list_N(M,A,n,Z) ==
      ∃zero[M]. empty(M,zero) &
                is_iterates(M, is_list_functor(M,A), zero, n, Z)"

definition
  mem_list :: "[i=>o,i,i] => o" where
    "mem_list(M,A,l) ==
      ∃n[M]. ∃listn[M].
       finite_ordinal(M,n) & is_list_N(M,A,n,listn) & l ∈ listn"

definition
  is_list :: "[i=>o,i,i] => o" where
    "is_list(M,A,Z) == ∀l[M]. l ∈ Z <-> mem_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) == (λX. ((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 ∈ formula ==> ∀n∈nat. depth(p) < n --> p ∈ formula_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 ∈ nat; p ∉ formula_N(n); p ∈ formula_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 ∈ nat; n ∈ nat |] ==> m≤n --> formula_N(m) ⊆ formula_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 ∈ nat; n ∈ nat |] ==> formula_N(m ∪ n) = formula_N(m) ∪ formula_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,
         λn 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,
```

```
        λn 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, λn y. n∈nat & y = (λX. {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)" _ "λX. {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, λn y. n∈nat & y = (λX. ((nat*nat) + (nat*nat))
+ (X*X + X))^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)" _
                                  "λX. ((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^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^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 ∈ 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 ∈ list(A) ==> ∀n ∈ nat. nth(n,xs) = hd' (tl'^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 ∈ nat; M(x)|] ==> M(tl'^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 ∈ list(A); n ∈ 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_nth(M,n,l,Z) ==
      ∃X[M]. is_iterates(M, is_tl(M), l, n, X) & is_hd(M,X,Z)"

lemma (in M_datatypes) nth_abs [simp]:
     "[|M(A); n ∈ nat; l ∈ list(A); M(Z)|]
      ==> is_nth(M,n,l,Z) <-> 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,
                λu v. c(u, v, h ‘ succ(depth(u)) ‘ u,
                              h ‘ succ(depth(v)) ‘ v),
                λu. d(u, h ‘ succ(depth(u)) ‘ u))"
```

Unfold *formula_rec* to *formula_rec_case*. Express *formula_rec* without using *rank* or *λx. Vset(x)*, 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)),
             λx 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 add: formula_case_abs)
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_c(f),
             λu v. c(u, v, f ` succ(depth(u)) ` u, f ` succ(depth(v))
` v))"
      and d_closed: "[|x ∈ formula; M(gx)|] ==> M(d(x, gx))"
      and d_rel:
         "M(f) ==>
```

```
         Relation1(M, formula, is_d(f), λu. d(u, f ` succ(depth(u)) `
u))"
      and fr_replace: "n ∈ nat ==> transrec_replacement(M,MH,n)"
      and fr_lam_replace:
           "M(g) ==>
            strong_replacement
              (M, λx y. x ∈ formula &
                  y = ⟨x, formula_rec_case(a,b,c,d,g,x)⟩)"

lemma (in Formula_Rec) formula_rec_case_closed:
    "[|M(g); p ∈ formula|] ==> M(formula_rec_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_rec_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,
             λx h. Lambda (formula, formula_rec_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 ∈ nat
     ==> M(transrec
          (n, λx h. Lambda(formula, formula_rec_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 ∈ formula ==> M(formula_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 ∈ formula; M(z)|]
   ==> is_formula_rec(M,MH,p,z) <-> z = formula_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
```

7 Closed Unbounded Classes and Normal Functions

theory *Normal* **imports** *Main* **begin**

One source is the book

Frank R. Drake. *Set Theory: An Introduction to Large Cardinals.* North-Holland, 1974.

7.1 Closed and Unbounded (c.u.) Classes of Ordinals

```
definition
  Closed :: "(i=>o) => o" where
    "Closed(P) == ∀I. I ≠ 0 --> (∀i∈I. Ord(i) ∧ P(i)) --> P(⋃(I))"
```

```
definition
  Unbounded :: "(i=>o) => o" where
    "Unbounded(P) == ∀i. Ord(i) --> (∃j. i<j ∧ P(j))"
```

```
definition
  Closed_Unbounded :: "(i=>o) => o" where
    "Closed_Unbounded(P) == Closed(P) ∧ Unbounded(P)"
```

7.1.1 Simple facts about c.u. classes

```
lemma ClosedI:
     "[| !!I. [| I ≠ 0; ∀i∈I. Ord(i) ∧ P(i) |] ==> P(⋃(I)) |]
      ==> Closed(P)"
by (simp add: Closed_def)
```

```
lemma ClosedD:
     "[| Closed(P); I ≠ 0; !!i. i∈I ==> Ord(i); !!i. i∈I ==> P(i) |]

      ==> P(⋃(I))"
by (simp add: Closed_def)
```

```
lemma UnboundedD:
     "[| Unbounded(P);  Ord(i) |] ==> ∃j. i<j ∧ P(j)"
by (simp add: Unbounded_def)
```

```
lemma Closed_Unbounded_imp_Unbounded: "Closed_Unbounded(C) ==> Unbounded(C)"
by (simp add: Closed_Unbounded_def)
```

The universal class, V, is closed and unbounded. A bit odd, since C. U. concerns only ordinals, but it's used below!

```
theorem Closed_Unbounded_V [simp]: "Closed_Unbounded(λx. True)"
by (unfold Closed_Unbounded_def Closed_def Unbounded_def, blast)
```

The class of ordinals, *Ord*, is closed and unbounded.

```
theorem Closed_Unbounded_Ord   [simp]: "Closed_Unbounded(Ord)"
by (unfold Closed_Unbounded_def Closed_def Unbounded_def, blast)
```

The class of limit ordinals, `Limit`, is closed and unbounded.

```
theorem Closed_Unbounded_Limit [simp]: "Closed_Unbounded(Limit)"
apply (simp add: Closed_Unbounded_def Closed_def Unbounded_def Limit_Union,

       clarify)
apply (rule_tac x="i++nat" in exI)
apply (blast intro: oadd_lt_self oadd_LimitI Limit_nat Limit_has_0)
done
```

The class of cardinals, `Card`, is closed and unbounded.

```
theorem Closed_Unbounded_Card  [simp]: "Closed_Unbounded(Card)"
apply (simp add: Closed_Unbounded_def Closed_def Unbounded_def Card_Union)
apply (blast intro: lt_csucc Card_csucc)
done
```

7.1.2 The intersection of any set-indexed family of c.u. classes is c.u.

The constructions below come from Kunen, *Set Theory*, page 78.

```
locale cub_family =
  fixes P and A
  fixes next_greater — the next ordinal satisfying class A
  fixes sup_greater  — sup of those ordinals over all A
  assumes closed:     "a∈A ==> Closed(P(a))"
      and unbounded: "a∈A ==> Unbounded(P(a))"
      and A_non0: "A≠0"
  defines "next_greater(a,x) == μ y. x<y ∧ P(a,y)"
      and "sup_greater(x) == ⋃a∈A. next_greater(a,x)"
```

Trivial that the intersection is closed.

```
lemma (in cub_family) Closed_INT: "Closed(λx. ∀i∈A. P(i,x))"
by (blast intro: ClosedI ClosedD [OF closed])
```

All remaining effort goes to show that the intersection is unbounded.

```
lemma (in cub_family) Ord_sup_greater:
     "Ord(sup_greater(x))"
by (simp add: sup_greater_def next_greater_def)

lemma (in cub_family) Ord_next_greater:
     "Ord(next_greater(a,x))"
by (simp add: next_greater_def Ord_Least)
```

`next_greater` works as expected: it returns a larger value and one that belongs to class `P(a)`.

```
lemma (in cub_family) next_greater_lemma:
     "[| Ord(x); a∈A |] ==> P(a, next_greater(a,x)) ∧ x < next_greater(a,x)"
apply (simp add: next_greater_def)
apply (rule exE [OF UnboundedD [OF unbounded]])
  apply assumption+
apply (blast intro: LeastI2 lt_Ord2)
done

lemma (in cub_family) next_greater_in_P:
     "[| Ord(x); a∈A |] ==> P(a, next_greater(a,x))"
by (blast dest: next_greater_lemma)

lemma (in cub_family) next_greater_gt:
     "[| Ord(x); a∈A |] ==> x < next_greater(a,x)"
by (blast dest: next_greater_lemma)

lemma (in cub_family) sup_greater_gt:
     "Ord(x) ==> x < sup_greater(x)"
apply (simp add: sup_greater_def)
apply (insert A_non0)
apply (blast intro: UN_upper_lt next_greater_gt Ord_next_greater)
done

lemma (in cub_family) next_greater_le_sup_greater:
     "a∈A ==> next_greater(a,x) ≤ sup_greater(x)"
apply (simp add: sup_greater_def)
apply (blast intro: UN_upper_le Ord_next_greater)
done

lemma (in cub_family) omega_sup_greater_eq_UN:
     "[| Ord(x); a∈A |]
      ==> sup_greater^ω (x) =
          (⋃n∈nat. next_greater(a, sup_greater^n (x)))"
apply (simp add: iterates_omega_def)
apply (rule le_anti_sym)
apply (rule le_implies_UN_le_UN)
apply (blast intro: leI next_greater_gt Ord_iterates Ord_sup_greater)
```

Opposite bound:

```
 1. ⟦Ord(x); a ∈ A⟧
    ⟹ (⋃n∈nat. next_greater(a, sup_greater^n (x))) ≤
        (⋃n∈nat. sup_greater^n (x))
```

```
apply (rule UN_least_le)
apply (blast intro: Ord_UN Ord_iterates Ord_sup_greater)
apply (rule_tac a="succ(n)" in UN_upper_le)
apply (simp_all add: next_greater_le_sup_greater)
apply (blast intro: Ord_UN Ord_iterates Ord_sup_greater)
done
```

```
lemma (in cub_family) P_omega_sup_greater:
     "[| Ord(x); a∈A |] ==> P(a, sup_greater^ω (x))"
apply (simp add: omega_sup_greater_eq_UN)
apply (rule ClosedD [OF closed])
apply (blast intro: ltD, auto)
apply (blast intro: Ord_iterates Ord_next_greater Ord_sup_greater)
apply (blast intro: next_greater_in_P Ord_iterates Ord_sup_greater)
done

lemma (in cub_family) omega_sup_greater_gt:
     "Ord(x) ==> x < sup_greater^ω (x)"
apply (simp add: iterates_omega_def)
apply (rule UN_upper_lt [of 1], simp_all)
 apply (blast intro: sup_greater_gt)
apply (blast intro: Ord_UN Ord_iterates Ord_sup_greater)
done

lemma (in cub_family) Unbounded_INT: "Unbounded(λx. ∀a∈A. P(a,x))"
apply (unfold Unbounded_def)
apply (blast intro!: omega_sup_greater_gt P_omega_sup_greater)
done

lemma (in cub_family) Closed_Unbounded_INT:
     "Closed_Unbounded(λx. ∀a∈A. P(a,x))"
by (simp add: Closed_Unbounded_def Closed_INT Unbounded_INT)


theorem Closed_Unbounded_INT:
    "(!!a. a∈A ==> Closed_Unbounded(P(a)))
     ==> Closed_Unbounded(λx. ∀a∈A. P(a, x))"
apply (case_tac "A=0", simp)
apply (rule cub_family.Closed_Unbounded_INT [OF cub_family.intro])
apply (simp_all add: Closed_Unbounded_def)
done

lemma Int_iff_INT2:
     "P(x) ∧ Q(x)  <->  (∀i∈2. (i=0 --> P(x)) ∧ (i=1 --> Q(x)))"
by auto

theorem Closed_Unbounded_Int:
     "[| Closed_Unbounded(P); Closed_Unbounded(Q) |]
      ==> Closed_Unbounded(λx. P(x) ∧ Q(x))"
apply (simp only: Int_iff_INT2)
apply (rule Closed_Unbounded_INT, auto)
done
```

7.2 Normal Functions

```
definition
  mono_le_subset :: "(i=>i) => o" where
    "mono_le_subset(M) == ∀i j. i≤j --> M(i) ⊆ M(j)"

definition
  mono_Ord :: "(i=>i) => o" where
    "mono_Ord(F) == ∀i j. i<j --> F(i) < F(j)"

definition
  cont_Ord :: "(i=>i) => o" where
    "cont_Ord(F) == ∀l. Limit(l) --> F(l) = (⋃i<l. F(i))"

definition
  Normal :: "(i=>i) => o" where
    "Normal(F) == mono_Ord(F) ∧ cont_Ord(F)"
```

7.2.1 Immediate properties of the definitions

```
lemma NormalI:
     "[|!!i j. i<j ==> F(i) < F(j);  !!l. Limit(l) ==> F(l) = (⋃i<l.
F(i))|]
      ==> Normal(F)"
by (simp add: Normal_def mono_Ord_def cont_Ord_def)

lemma mono_Ord_imp_Ord: "[| Ord(i); mono_Ord(F) |] ==> Ord(F(i))"
apply (simp add: mono_Ord_def)
apply (blast intro: lt_Ord)
done

lemma mono_Ord_imp_mono: "[| i<j; mono_Ord(F) |] ==> F(i) < F(j)"
by (simp add: mono_Ord_def)

lemma Normal_imp_Ord [simp]: "[| Normal(F); Ord(i) |] ==> Ord(F(i))"
by (simp add: Normal_def mono_Ord_imp_Ord)

lemma Normal_imp_cont: "[| Normal(F); Limit(l) |] ==> F(l) = (⋃i<l.
F(i))"
by (simp add: Normal_def cont_Ord_def)

lemma Normal_imp_mono: "[| i<j; Normal(F) |] ==> F(i) < F(j)"
by (simp add: Normal_def mono_Ord_def)

lemma Normal_increasing: "[| Ord(i); Normal(F) |] ==> i≤F(i)"
apply (induct i rule: trans_induct3_rule)
  apply (simp add: subset_imp_le)
 apply (subgoal_tac "F(x) < F(succ(x))")
  apply (force intro: lt_trans1)
 apply (simp add: Normal_def mono_Ord_def)
```

```
apply (subgoal_tac "(⋃y<x. y) ≤ (⋃y<x. F(y))")
 apply (simp add: Normal_imp_cont Limit_OUN_eq)
apply (blast intro: ltD le_implies_OUN_le_OUN)
done
```

7.2.2 The class of fixedpoints is closed and unbounded

The proof is from Drake, pages 113–114.

```
lemma mono_Ord_imp_le_subset: "mono_Ord(F) ==> mono_le_subset(F)"
apply (simp add: mono_le_subset_def, clarify)
apply (subgoal_tac "F(i)≤F(j)", blast dest: le_imp_subset)
apply (simp add: le_iff)
apply (blast intro: lt_Ord2 mono_Ord_imp_Ord mono_Ord_imp_mono)
done
```

The following equation is taken for granted in any set theory text.

```
lemma cont_Ord_Union:
     "[| cont_Ord(F); mono_le_subset(F); X=0 --> F(0)=0; ∀x∈X. Ord(x)
|]
      ==> F(Union(X)) = (⋃y∈X. F(y))"
apply (frule Ord_set_cases)
apply (erule disjE, force)
apply (thin_tac "X=0 --> ?Q", auto)
```

The trival case of $\bigcup X \in X$

```
 apply (rule equalityI, blast intro: Ord_Union_eq_succD)
 apply (simp add: mono_le_subset_def UN_subset_iff le_subset_iff)
 apply (blast elim: equalityE)
```

The limit case, `Limit(⋃X)`:

```
 1. ⟦cont_Ord(F); mono_le_subset(F); ∀x∈X. Ord(x); ⋃X ∉ X;
    Limit(⋃X)⟧
    ⟹ F(⋃X) = (⋃y∈X. F(y))
```

```
apply (simp add: OUN_Union_eq cont_Ord_def)
apply (rule equalityI)
```

First inclusion:

```
 apply (rule UN_least [OF OUN_least])
 apply (simp add: mono_le_subset_def, blast intro: leI)
```

Second inclusion:

```
apply (rule UN_least)
apply (frule Union_upper_le, blast, blast intro: Ord_Union)
apply (erule leE, drule ltD, elim UnionE)
 apply (simp add: OUnion_def)
 apply blast+
```

done

lemma *Normal_Union:*
```
     "[| X≠0; ∀x∈X. Ord(x); Normal(F) |] ==> F(Union(X)) = (⋃y∈X. F(y))"
```
apply *(simp add: Normal_def)*
apply *(blast intro: mono_Ord_imp_le_subset cont_Ord_Union)*
done

lemma *Normal_imp_fp_Closed: "Normal(F) ==> Closed(λi. F(i) = i)"*
apply *(simp add: Closed_def ball_conj_distrib, clarify)*
apply *(frule Ord_set_cases)*
apply *(auto simp add: Normal_Union)*
done

lemma *iterates_Normal_increasing:*
```
     "[| n∈nat;  x < F(x);  Normal(F) |]
      ==> F^n (x) < F^(succ(n)) (x)"
```
apply *(induct n rule: nat_induct)*
apply *(simp_all add: Normal_imp_mono)*
done

lemma *Ord_iterates_Normal:*
```
     "[| n∈nat;  Normal(F);  Ord(x) |] ==> Ord(F^n (x))"
```
by *(simp add: Ord_iterates)*

THIS RESULT IS UNUSED

lemma *iterates_omega_Limit:*
```
     "[| Normal(F);  x < F(x) |] ==> Limit(F^ω (x))"
```
apply *(frule lt_Ord)*
apply *(simp add: iterates_omega_def)*
apply *(rule increasing_LimitI)*
— this lemma is ⟦*0 < l; ∀x∈l. ∃y∈l. x < y*⟧ ⟹ *Limit(l)*
 apply *(blast intro: UN_upper_lt [of "1"]   Normal_imp_Ord*
Ord_UN Ord_iterates lt_imp_0_lt
iterates_Normal_increasing, clarify)
apply *(rule bexI)*
 apply *(blast intro: Ord_in_Ord [OF Ord_iterates_Normal])*
apply *(rule UN_I, erule nat_succI)*
apply *(blast intro: iterates_Normal_increasing Ord_iterates_Normal*
ltD [OF lt_trans1, OF succ_leI, OF ltI])
done

lemma *iterates_omega_fixedpoint:*
```
     "[| Normal(F); Ord(a) |] ==> F(F^ω (a)) = F^ω (a)"
```
apply *(frule Normal_increasing, assumption)*
apply *(erule leE)*
 apply *(simp_all add: iterates_omega_triv [OF sym])*
apply *(simp add: iterates_omega_def Normal_Union)*

apply *(rule equalityI, force simp add: nat_succI)*

Opposite inclusion:

```
 1. ⟦Normal(F); Ord(a); a < F(a)⟧
    ⟹ (⋃n∈nat. F^n (a)) ⊆ (⋃x∈nat. F(F^x (a)))
```

apply *clarify*
apply *(rule UN_I, assumption)*
apply *(frule iterates_Normal_increasing, assumption, assumption, simp)*
apply *(blast intro: Ord_trans ltD Ord_iterates_Normal Normal_imp_Ord [of F])*
done

lemma *iterates_omega_increasing:*
  "[| Normal(F); Ord(a) |] ==> a ≤ F^ω (a)"
apply *(unfold iterates_omega_def)*
apply *(rule UN_upper_le [of 0], simp_all)*
done

lemma *Normal_imp_fp_Unbounded: "Normal(F) ==> Unbounded(λi. F(i) = i)"*
apply *(unfold Unbounded_def, clarify)*
apply *(rule_tac x="F^ω (succ(i))"* **in** *exI)*
apply *(simp add: iterates_omega_fixedpoint)*
apply *(blast intro: lt_trans2 [OF _ iterates_omega_increasing])*
done

theorem *Normal_imp_fp_Closed_Unbounded:*
  "Normal(F) ==> Closed_Unbounded(λi. F(i) = i)"
by *(simp add: Closed_Unbounded_def Normal_imp_fp_Closed Normal_imp_fp_Unbounded)*

7.2.3 Function *normalize*

Function *normalize* maps a function *F* to a normal function that bounds it above. The result is normal if and only if *F* is continuous: succ is not bounded above by any normal function, by *Normal_imp_fp_Unbounded*.

definition
 normalize :: "[i=>i, i] => i" **where**
  "normalize(F,a) == transrec2(a, F(0), λx r. F(succ(x)) Un succ(r))"

lemma *Ord_normalize [simp, intro]:*
  "[| Ord(a); !!x. Ord(x) ==> Ord(F(x)) |] ==> Ord(normalize(F, a))"
apply *(induct a rule: trans_induct3_rule)*
apply *(simp_all add: ltD def_transrec2 [OF normalize_def])*
done

```
lemma normalize_lemma [rule_format]:
     "[| Ord(b); !!x. Ord(x) ==> Ord(F(x)) |]
      ==> ∀a. a < b --> normalize(F, a) < normalize(F, b)"
apply (erule trans_induct3)
  apply (simp_all add: le_iff def_transrec2 [OF normalize_def])
 apply clarify
apply (rule Un_upper2_lt)
  apply auto
 apply (drule spec, drule mp, assumption)
 apply (erule leI)
apply (drule Limit_has_succ, assumption)
apply (blast intro!: Ord_normalize intro: OUN_upper_lt ltD lt_Ord)
done

lemma normalize_increasing:
     "[| a < b;  !!x. Ord(x) ==> Ord(F(x)) |]
      ==> normalize(F, a) < normalize(F, b)"
by (blast intro!: normalize_lemma intro: lt_Ord2)

theorem Normal_normalize:
     "(!!x. Ord(x) ==> Ord(F(x))) ==> Normal(normalize(F))"
apply (rule NormalI)
apply (blast intro!: normalize_increasing)
apply (simp add: def_transrec2 [OF normalize_def])
done

theorem le_normalize:
     "[| Ord(a); cont_Ord(F); !!x. Ord(x) ==> Ord(F(x)) |]
      ==> F(a) ≤ normalize(F,a)"
apply (erule trans_induct3)
apply (simp_all add: def_transrec2 [OF normalize_def])
apply (simp add: Un_upper1_le)
apply (simp add: cont_Ord_def)
apply (blast intro: ltD le_implies_OUN_le_OUN)
done
```

7.3 The Alephs

This is the well-known transfinite enumeration of the cardinal numbers.

```
definition
  Aleph :: "i => i" where
    "Aleph(a) == transrec2(a, nat, λx r. csucc(r))"

notation (xsymbols)
  Aleph  ("ℵ_" [90] 90)

lemma Card_Aleph [simp, intro]:
     "Ord(a) ==> Card(Aleph(a))"
apply (erule trans_induct3)
```

```
apply (simp_all add: Card_csucc Card_nat Card_is_Ord
                      def_transrec2 [OF Aleph_def])
done

lemma Aleph_lemma [rule_format]:
     "Ord(b) ==> ∀a. a < b --> Aleph(a) < Aleph(b)"
apply (erule trans_induct3)
apply (simp_all add: le_iff def_transrec2 [OF Aleph_def])
apply (blast intro: lt_trans lt_csucc Card_is_Ord, clarify)
apply (drule Limit_has_succ, assumption)
apply (blast intro: Card_is_Ord Card_Aleph OUN_upper_lt ltD lt_Ord)
done

lemma Aleph_increasing:
     "a < b ==> Aleph(a) < Aleph(b)"
by (blast intro!: Aleph_lemma intro: lt_Ord2)

theorem Normal_Aleph: "Normal(Aleph)"
apply (rule NormalI)
apply (blast intro!: Aleph_increasing)
apply (simp add: def_transrec2 [OF Aleph_def])
done

end
```

8 The Reflection Theorem

```
theory Reflection imports Normal begin

lemma all_iff_not_ex_not: "(∀x. P(x)) <-> (~ (∃x. ~ P(x)))"
by blast

lemma ball_iff_not_bex_not: "(∀x∈A. P(x)) <-> (~ (∃x∈A. ~ P(x)))"
by blast
```

From the notes of A. S. Kechris, page 6, and from Andrzej Mostowski, *Constructible Sets with Applications*, North-Holland, 1969, page 23.

8.1 Basic Definitions

First part: the cumulative hierarchy defining the class `M`. To avoid handling multiple arguments, we assume that `Mset(l)` is closed under ordered pairing provided `l` is limit. Possibly this could be avoided: the induction hypothesis `Cl_reflects` (in locale `ex_reflection`) could be weakened to `∀y∈Mset(a). ∀z∈Mset(a). P(⟨y, z⟩) ⟷ Q(a, ⟨y, z⟩)`, removing most uses of `Pair_in_Mset`. But there isn't much point in doing so, since ultimately the `ex_reflection` proof is packaged up using the predicate `Reflects`.

```
locale reflection =
  fixes Mset and M and Reflects
  assumes Mset_mono_le : "mono_le_subset(Mset)"
      and Mset_cont    : "cont_Ord(Mset)"
      and Pair_in_Mset : "[| x ∈ Mset(a); y ∈ Mset(a); Limit(a) |]
                          ==> <x,y> ∈ Mset(a)"
  defines "M(x) == ∃a. Ord(a) & x ∈ Mset(a)"
      and "Reflects(Cl,P,Q) == Closed_Unbounded(Cl) &
                              (∀a. Cl(a) --> (∀x∈Mset(a). P(x) <-> Q(a,x)))"
  fixes F0 — ordinal for a specific value y
  fixes FF — sup over the whole level, y ∈ Mset(a)
  fixes ClEx — Reflecting ordinals for the formula ∃z. P
  defines "F0(P,y) == μ b. (∃z. M(z) & P(<y,z>)) -->
                               (∃z∈Mset(b). P(<y,z>))"
      and "FF(P)   == λa. ⋃y∈Mset(a). F0(P,y)"
      and "ClEx(P,a) == Limit(a) & normalize(FF(P),a) = a"

lemma (in reflection) Mset_mono: "i≤j ==> Mset(i) <= Mset(j)"
apply (insert Mset_mono_le)
apply (simp add: mono_le_subset_def leI)
done
```

Awkward: we need a version of `ClEx_def` as an equality at the level of classes, which do not really exist

```
lemma (in reflection) ClEx_eq:
     "ClEx(P) == λa. Limit(a) & normalize(FF(P),a) = a"
by (simp add: ClEx_def [symmetric])
```

8.2 Easy Cases of the Reflection Theorem

```
theorem (in reflection) Triv_reflection [intro]:
     "Reflects(Ord, P, λa x. P(x))"
by (simp add: Reflects_def)

theorem (in reflection) Not_reflection [intro]:
     "Reflects(Cl,P,Q) ==> Reflects(Cl, λx. ~P(x), λa x. ~Q(a,x))"
by (simp add: Reflects_def)

theorem (in reflection) And_reflection [intro]:
     "[| Reflects(Cl,P,Q); Reflects(C',P',Q') |]
      ==> Reflects(λa. Cl(a) & C'(a), λx. P(x) & P'(x),
                                      λa x. Q(a,x) & Q'(a,x))"
by (simp add: Reflects_def Closed_Unbounded_Int, blast)

theorem (in reflection) Or_reflection [intro]:
     "[| Reflects(Cl,P,Q); Reflects(C',P',Q') |]
      ==> Reflects(λa. Cl(a) & C'(a), λx. P(x) | P'(x),
                                      λa x. Q(a,x) | Q'(a,x))"
by (simp add: Reflects_def Closed_Unbounded_Int, blast)
```

```
theorem (in reflection) Imp_reflection [intro]:
     "[| Reflects(Cl,P,Q); Reflects(C',P',Q') |]
      ==> Reflects(λa. Cl(a) & C'(a),
                   λx. P(x) --> P'(x),
                   λa x. Q(a,x) --> Q'(a,x))"
by (simp add: Reflects_def Closed_Unbounded_Int, blast)

theorem (in reflection) Iff_reflection [intro]:
     "[| Reflects(Cl,P,Q); Reflects(C',P',Q') |]
      ==> Reflects(λa. Cl(a) & C'(a),
                   λx. P(x) <-> P'(x),
                   λa x. Q(a,x) <-> Q'(a,x))"
by (simp add: Reflects_def Closed_Unbounded_Int, blast)
```

8.3 Reflection for Existential Quantifiers

```
lemma (in reflection) F0_works:
     "[| y∈Mset(a); Ord(a); M(z); P(<y,z>) |] ==> ∃z∈Mset(F0(P,y)).
P(<y,z>)"
apply (unfold F0_def M_def, clarify)
apply (rule LeastI2)
  apply (blast intro: Mset_mono [THEN subsetD])
 apply (blast intro: lt_Ord2, blast)
done

lemma (in reflection) Ord_F0 [intro,simp]: "Ord(F0(P,y))"
by (simp add: F0_def)

lemma (in reflection) Ord_FF [intro,simp]: "Ord(FF(P,y))"
by (simp add: FF_def)

lemma (in reflection) cont_Ord_FF: "cont_Ord(FF(P))"
apply (insert Mset_cont)
apply (simp add: cont_Ord_def FF_def, blast)
done
```

Recall that `F0` depends upon `y ∈ Mset(a)`, while `FF` depends only upon `a`.

```
lemma (in reflection) FF_works:
     "[| M(z); y∈Mset(a); P(<y,z>); Ord(a) |] ==> ∃z∈Mset(FF(P,a)).
P(<y,z>)"
apply (simp add: FF_def)
apply (simp_all add: cont_Ord_Union [of concl: Mset]
                     Mset_cont Mset_mono_le not_emptyI Ord_F0)
apply (blast intro: F0_works)
done

lemma (in reflection) FFN_works:
     "[| M(z); y∈Mset(a); P(<y,z>); Ord(a) |]
```

```
      ==> ∃z∈Mset(normalize(FF(P),a)). P(<y,z>)"
apply (drule FF_works [of concl: P], assumption+)
apply (blast intro: cont_Ord_FF le_normalize [THEN Mset_mono, THEN subsetD])
done
```

Locale for the induction hypothesis

```
locale ex_reflection = reflection +
  fixes P   — the original formula
  fixes Q   — the reflected formula
  fixes Cl  — the class of reflecting ordinals
  assumes Cl_reflects: "[| Cl(a); Ord(a) |] ==> ∀x∈Mset(a). P(x) <->
Q(a,x)"

lemma (in ex_reflection) ClEx_downward:
     "[| M(z); y∈Mset(a); P(<y,z>); Cl(a); ClEx(P,a) |]
      ==> ∃z∈Mset(a). Q(a,<y,z>)"
apply (simp add: ClEx_def, clarify)
apply (frule Limit_is_Ord)
apply (frule FFN_works [of concl: P], assumption+)
apply (drule Cl_reflects, assumption+)
apply (auto simp add: Limit_is_Ord Pair_in_Mset)
done

lemma (in ex_reflection) ClEx_upward:
     "[| z∈Mset(a); y∈Mset(a); Q(a,<y,z>); Cl(a); ClEx(P,a) |]
      ==> ∃z. M(z) & P(<y,z>)"
apply (simp add: ClEx_def M_def)
apply (blast dest: Cl_reflects
             intro: Limit_is_Ord Pair_in_Mset)
done
```

Class `ClEx` indeed consists of reflecting ordinals...

```
lemma (in ex_reflection) ZF_ClEx_iff:
     "[| y∈Mset(a); Cl(a); ClEx(P,a) |]
      ==> (∃z. M(z) & P(<y,z>)) <-> (∃z∈Mset(a). Q(a,<y,z>))"
by (blast intro: dest: ClEx_downward ClEx_upward)
```

...and it is closed and unbounded

```
lemma (in ex_reflection) ZF_Closed_Unbounded_ClEx:
     "Closed_Unbounded(ClEx(P))"
apply (simp add: ClEx_eq)
apply (fast intro: Closed_Unbounded_Int Normal_imp_fp_Closed_Unbounded
                   Closed_Unbounded_Limit Normal_normalize)
done
```

The same two theorems, exported to locale `reflection`.

Class `ClEx` indeed consists of reflecting ordinals...

```
lemma (in reflection) ClEx_iff:
```

```
     "[| y∈Mset(a); Cl(a); ClEx(P,a);
        !!a. [| Cl(a); Ord(a) |] ==> ∀x∈Mset(a). P(x) <-> Q(a,x) |]
      ==> (∃z. M(z) & P(<y,z>)) <-> (∃z∈Mset(a). Q(a,<y,z>))"
apply (unfold ClEx_def FF_def F0_def M_def)
apply (rule ex_reflection.ZF_ClEx_iff
  [OF ex_reflection.intro, OF reflection.intro ex_reflection_axioms.intro,
    of Mset Cl])
apply (simp_all add: Mset_mono_le Mset_cont Pair_in_Mset)
done
```

```
lemma (in reflection) Closed_Unbounded_ClEx:
     "(!!a. [| Cl(a); Ord(a) |] ==> ∀x∈Mset(a). P(x) <-> Q(a,x))
      ==> Closed_Unbounded(ClEx(P))"
apply (unfold ClEx_eq FF_def F0_def M_def)
apply (rule ex_reflection.ZF_Closed_Unbounded_ClEx [of Mset _ _ Cl])
apply (rule ex_reflection.intro, rule reflection_axioms)
apply (blast intro: ex_reflection_axioms.intro)
done
```

8.4 Packaging the Quantifier Reflection Rules

```
lemma (in reflection) Ex_reflection_0:
     "Reflects(Cl,P0,Q0)
      ==> Reflects(λa. Cl(a) & ClEx(P0,a),
                   λx. ∃z. M(z) & P0(<x,z>),
                   λa x. ∃z∈Mset(a). Q0(a,<x,z>))"
apply (simp add: Reflects_def)
apply (intro conjI Closed_Unbounded_Int)
  apply blast
 apply (rule Closed_Unbounded_ClEx [of Cl P0 Q0], blast, clarify)
apply (rule_tac Cl=Cl in  ClEx_iff, assumption+, blast)
done
```

```
lemma (in reflection) All_reflection_0:
     "Reflects(Cl,P0,Q0)
      ==> Reflects(λa. Cl(a) & ClEx(λx.~P0(x), a),
                   λx. ∀z. M(z) --> P0(<x,z>),
                   λa x. ∀z∈Mset(a). Q0(a,<x,z>))"
apply (simp only: all_iff_not_ex_not ball_iff_not_bex_not)
apply (rule Not_reflection, drule Not_reflection, simp)
apply (erule Ex_reflection_0)
done
```

```
theorem (in reflection) Ex_reflection [intro]:
     "Reflects(Cl, λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x)))
      ==> Reflects(λa. Cl(a) & ClEx(λx. P(fst(x),snd(x)), a),
                   λx. ∃z. M(z) & P(x,z),
```

```
                  λa x. ∃z∈Mset(a). Q(a,x,z))"
by (rule Ex_reflection_0 [of _ " λx. P(fst(x),snd(x))"
                               "λa x. Q(a,fst(x),snd(x))", simplified])

theorem (in reflection) All_reflection [intro]:
     "Reflects(Cl,  λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x)))
      ==> Reflects(λa. Cl(a) & ClEx(λx. ~P(fst(x),snd(x)), a),
                  λx. ∀z. M(z) --> P(x,z),
                  λa x. ∀z∈Mset(a). Q(a,x,z))"
by (rule All_reflection_0 [of _ "λx. P(fst(x),snd(x))"
                                "λa x. Q(a,fst(x),snd(x))", simplified])
```

And again, this time using class-bounded quantifiers

```
theorem (in reflection) Rex_reflection [intro]:
     "Reflects(Cl, λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x)))
      ==> Reflects(λa. Cl(a) & ClEx(λx. P(fst(x),snd(x)), a),
                  λx. ∃z[M]. P(x,z),
                  λa x. ∃z∈Mset(a). Q(a,x,z))"
by (unfold rex_def, blast)

theorem (in reflection) Rall_reflection [intro]:
     "Reflects(Cl,  λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x)))
      ==> Reflects(λa. Cl(a) & ClEx(λx. ~P(fst(x),snd(x)), a),
                  λx. ∀z[M]. P(x,z),
                  λa x. ∀z∈Mset(a). Q(a,x,z))"
by (unfold rall_def, blast)
```

No point considering bounded quantifiers, where reflection is trivial.

8.5 Simple Examples of Reflection

Example 1: reflecting a simple formula. The reflecting class is first given as the variable `?Cl` and later retrieved from the final proof state.

```
lemma (in reflection)
     "Reflects(?Cl,
              λx. ∃y. M(y) & x ∈ y,
              λa x. ∃y∈Mset(a). x ∈ y)"
by fast
```

Problem here: there needs to be a conjunction (class intersection) in the class of reflecting ordinals. The `Ord(a)` is redundant, though harmless.

```
lemma (in reflection)
     "Reflects(λa. Ord(a) & ClEx(λx. fst(x) ∈ snd(x), a),
              λx. ∃y. M(y) & x ∈ y,
              λa x. ∃y∈Mset(a). x ∈ y)"
by fast
```

Example 2

```
lemma (in reflection)
     "Reflects(?Cl,
               λx. ∃y. M(y) & (∀z. M(z) --> z ⊆ x --> z ∈ y),
               λa x. ∃y∈Mset(a). ∀z∈Mset(a). z ⊆ x --> z ∈ y)"
by fast
```

Example 2'. We give the reflecting class explicitly.

```
lemma (in reflection)
  "Reflects
    (λa. (Ord(a) &
          ClEx(λx. ~ (snd(x) ⊆ fst(fst(x)) --> snd(x) ∈ snd(fst(x))),
a)) &
          ClEx(λx. ∀z. M(z) --> z ⊆ fst(x) --> z ∈ snd(x), a),
            λx. ∃y. M(y) & (∀z. M(z) --> z ⊆ x --> z ∈ y),
            λa x. ∃y∈Mset(a). ∀z∈Mset(a). z ⊆ x --> z ∈ y)"
by fast
```

Example 2". We expand the subset relation.

```
lemma (in reflection)
  "Reflects(?Cl,
        λx. ∃y. M(y) & (∀z. M(z) --> (∀w. M(w) --> w∈z --> w∈x) -->
z∈y),
        λa x. ∃y∈Mset(a). ∀z∈Mset(a). (∀w∈Mset(a). w∈z --> w∈x) -->
z∈y)"
by fast
```

Example 2"'. Single-step version, to reveal the reflecting class.

```
lemma (in reflection)
     "Reflects(?Cl,
               λx. ∃y. M(y) & (∀z. M(z) --> z ⊆ x --> z ∈ y),
               λa x. ∃y∈Mset(a). ∀z∈Mset(a). z ⊆ x --> z ∈ y)"
apply (rule Ex_reflection)


TERM λa. ?Cl3(a) ∧
         ClEx(λx. ∀z. M(z) ⟶ z ⊆ fst(x) ⟶ z ∈ snd(x),
              a) &&
Reflects
 (λa. ?Cl3(a) ∧
      ClEx(λx. ∀z. M(z) ⟶ z ⊆ fst(x) ⟶ z ∈ snd(x), a),
  λx. ∃y. M(y) ∧ (∀z. M(z) ⟶ z ⊆ x ⟶ z ∈ y),
  λa x. ∃y∈Mset(a). ∀z∈Mset(a). z ⊆ x ⟶ z ∈ y)
 1. Reflects
     (?Cl3, λx. ∀z. M(z) ⟶ z ⊆ fst(x) ⟶ z ∈ snd(x),
      λa x. ∀z∈Mset(a). z ⊆ fst(x) ⟶ z ∈ snd(x))


apply (rule All_reflection)
```

```
TERM λa. (?Cl5(a) ∧
          ClEx(λx. ¬ (snd(x) ⊆ fst(fst(x)) ⟶
                      snd(x) ∈ snd(fst(x))),
               a)) ∧
         ClEx(λx. ∀z. M(z) ⟶ z ⊆ fst(x) ⟶ z ∈ snd(x),
              a) &&
Reflects
 (λa. (?Cl5(a) ∧
       ClEx(λx. ¬ (snd(x) ⊆ fst(fst(x)) ⟶
                   snd(x) ∈ snd(fst(x))),
            a)) ∧
      ClEx(λx. ∀z. M(z) ⟶ z ⊆ fst(x) ⟶ z ∈ snd(x), a),
  λx. ∃y. M(y) ∧ (∀z. M(z) ⟶ z ⊆ x ⟶ z ∈ y),
  λa x. ∃y∈Mset(a). ∀z∈Mset(a). z ⊆ x ⟶ z ∈ y)
 1. Reflects
     (?Cl5,
      λx. snd(x) ⊆ fst(fst(x)) ⟶ snd(x) ∈ snd(fst(x)),
      λa x. snd(x) ⊆ fst(fst(x)) ⟶ snd(x) ∈ snd(fst(x)))
```

apply *(rule Triv_reflection)*

```
TERM λa. (Ord(a) ∧
          ClEx(λx. ¬ (snd(x) ⊆ fst(fst(x)) ⟶
                      snd(x) ∈ snd(fst(x))),
               a)) ∧
         ClEx(λx. ∀z. M(z) ⟶ z ⊆ fst(x) ⟶ z ∈ snd(x),
              a) &&
Reflects
 (λa. (Ord(a) ∧
       ClEx(λx. ¬ (snd(x) ⊆ fst(fst(x)) ⟶
                   snd(x) ∈ snd(fst(x))),
            a)) ∧
      ClEx(λx. ∀z. M(z) ⟶ z ⊆ fst(x) ⟶ z ∈ snd(x), a),
  λx. ∃y. M(y) ∧ (∀z. M(z) ⟶ z ⊆ x ⟶ z ∈ y),
  λa x. ∃y∈Mset(a). ∀z∈Mset(a). z ⊆ x ⟶ z ∈ y)
No subgoals!
```

done

Example 3. Warning: the following examples make sense only if *P* is quantifier-free, since it is not being relativized.

lemma *(***in** *reflection)*

```
    "Reflects(?Cl,
              λx. ∃y. M(y) & (∀z. M(z) --> z ∈ y <-> z ∈ x & P(z)),

              λa x. ∃y∈Mset(a). ∀z∈Mset(a). z ∈ y <-> z ∈ x & P(z))"
```

by *fast*

Example 3'

```
lemma (in reflection)
     "Reflects(?Cl,
               λx. ∃y. M(y) & y = Collect(x,P),
               λa x. ∃y∈Mset(a). y = Collect(x,P))"
by fast
```

Example 3"

```
lemma (in reflection)
     "Reflects(?Cl,
               λx. ∃y. M(y) & y = Replace(x,P),
               λa x. ∃y∈Mset(a). y = Replace(x,P))"
by fast
```

Example 4: Axiom of Choice. Possibly wrong, since Π needs to be relativized.

```
lemma (in reflection)
     "Reflects(?Cl,
               λA. 0∉A --> (∃f. M(f) & f ∈ (Π X ∈ A. X)),
               λa A. 0∉A --> (∃f∈Mset(a). f ∈ (Π X ∈ A. X)))"
by fast

end
```

9 The meta-existential quantifier

```
theory MetaExists imports Main begin
```

Allows quantification over any term having sort `logic`. Used to quantify over classes. Yields a proposition rather than a FOL formula.

```
definition
  ex :: "(('a::{}) => prop) => prop"  (binder "?? " 0) where
  "ex(P) == (!!Q. (!!x. PROP P(x) ==> PROP Q) ==> PROP Q)"

notation (xsymbols)
  ex  (binder "⋁" 0)

lemma meta_exI: "PROP P(x) ==> (?? x. PROP P(x))"
proof (unfold ex_def)
  assume P: "PROP P(x)"
  fix Q
  assume PQ: "⋀x. PROP P(x) ⟹ PROP Q"
  from P show "PROP Q" by (rule PQ)
qed
```

```
lemma meta_exE: "[| ?? x. PROP P(x);  !!x. PROP P(x) ==> PROP R |] ==>
PROP R"
proof (unfold ex_def)
  assume QPQ: "⋀Q. (⋀x. PROP P(x) ⟹ PROP Q) ⟹ PROP Q"
  assume PR: "⋀x. PROP P(x) ⟹ PROP R"
  from PR show "PROP R" by (rule QPQ)
qed

end
```

10 The ZF Axioms (Except Separation) in L

```
theory L_axioms imports Formula Relative Reflection MetaExists begin
```

The class L satisfies the premises of locale `M_trivial`

```
lemma transL: "[| y∈x; L(x) |] ==> L(y)"
apply (insert Transset_Lset)
apply (simp add: Transset_def L_def, blast)
done

lemma nonempty: "L(0)"
apply (simp add: L_def)
apply (blast intro: zero_in_Lset)
done

theorem upair_ax: "upair_ax(L)"
apply (simp add: upair_ax_def upair_def, clarify)
apply (rule_tac x="{x,y}" in rexI)
apply (simp_all add: doubleton_in_L)
done

theorem Union_ax: "Union_ax(L)"
apply (simp add: Union_ax_def big_union_def, clarify)
apply (rule_tac x="Union(x)" in rexI)
apply (simp_all add: Union_in_L, auto)
apply (blast intro: transL)
done

theorem power_ax: "power_ax(L)"
apply (simp add: power_ax_def powerset_def Relative.subset_def, clarify)
apply (rule_tac x="{y ∈ Pow(x). L(y)}" in rexI)
apply (simp_all add: LPow_in_L, auto)
apply (blast intro: transL)
done
```

We don't actually need `L` to satisfy the foundation axiom.

```
theorem foundation_ax: "foundation_ax(L)"
```

```
apply (simp add: foundation_ax_def)
apply (rule rallI)
apply (cut_tac A=x in foundation)
apply (blast intro: transL)
done
```

10.1 For L to satisfy Replacement

```
lemma LReplace_in_Lset:
     "[|X ∈ Lset(i); univalent(L,X,Q); Ord(i)|]
      ==> ∃j. Ord(j) & Replace(X, %x y. Q(x,y) & L(y)) ⊆ Lset(j)"
apply (rule_tac x="⋃y ∈ Replace(X, %x y. Q(x,y) & L(y)). succ(lrank(y))"
       in exI)
apply simp
apply clarify
apply (rule_tac a=x in UN_I)
 apply (simp_all add: Replace_iff univalent_def)
apply (blast dest: transL L_I)
done

lemma LReplace_in_L:
     "[|L(X); univalent(L,X,Q)|]
      ==> ∃Y. L(Y) & Replace(X, %x y. Q(x,y) & L(y)) ⊆ Y"
apply (drule L_D, clarify)
apply (drule LReplace_in_Lset, assumption+)
apply (blast intro: L_I Lset_in_Lset_succ)
done

theorem replacement: "replacement(L,P)"
apply (simp add: replacement_def, clarify)
apply (frule LReplace_in_L, assumption+, clarify)
apply (rule_tac x=Y in rexI)
apply (simp_all add: Replace_iff univalent_def, blast)
done
```

10.2 Instantiating the locale M_trivial

No instances of Separation yet.

```
lemma Lset_mono_le: "mono_le_subset(Lset)"
by (simp add: mono_le_subset_def le_imp_subset Lset_mono)

lemma Lset_cont: "cont_Ord(Lset)"
by (simp add: cont_Ord_def Limit_Lset_eq OUnion_def Limit_is_Ord)

lemmas L_nat = Ord_in_L [OF Ord_nat]

theorem M_trivial_L: "PROP M_trivial(L)"
  apply (rule M_trivial.intro)
       apply (erule (1) transL)
```

```
      apply (rule upair_ax)
     apply (rule Union_ax)
    apply (rule power_ax)
   apply (rule replacement)
  apply (rule L_nat)
  done
```

```
interpretation M_trivial ["L"] by (rule M_trivial_L)
```

10.3 Instantiation of the locale *reflection*

instances of locale constants

```
definition
  L_F0 :: "[i=>o,i] => i" where
    "L_F0(P,y) == μ b. (∃z. L(z) ∧ P(<y,z>)) --> (∃z∈Lset(b). P(<y,z>))"
```

```
definition
  L_FF :: "[i=>o,i] => i" where
    "L_FF(P)   == λa. ⋃y∈Lset(a). L_F0(P,y)"
```

```
definition
  L_ClEx :: "[i=>o,i] => o" where
    "L_ClEx(P) == λa. Limit(a) ∧ normalize(L_FF(P),a) = a"
```

We must use the meta-existential quantifier; otherwise the reflection terms become enormous!

```
definition
  L_Reflects :: "[i=>o,[i,i]=>o] => prop"  ("(3REFLECTS/ [_,/ _])") where
    "REFLECTS[P,Q] == (??Cl. Closed_Unbounded(Cl) &
                            (∀a. Cl(a) --> (∀x ∈ Lset(a). P(x) <-> Q(a,x))))"
```

```
theorem Triv_reflection:
     "REFLECTS[P, λa x. P(x)]"
apply (simp add: L_Reflects_def)
apply (rule meta_exI)
apply (rule Closed_Unbounded_Ord)
done
```

```
theorem Not_reflection:
     "REFLECTS[P,Q] ==> REFLECTS[λx. ~P(x), λa x. ~Q(a,x)]"
apply (unfold L_Reflects_def)
apply (erule meta_exE)
apply (rule_tac x=Cl in meta_exI, simp)
done
```

```
theorem And_reflection:
     "[| REFLECTS[P,Q]; REFLECTS[P',Q'] |]
```

```
      ==> REFLECTS[λx. P(x) ∧ P'(x), λa x. Q(a,x) ∧ Q'(a,x)]"
apply (unfold L_Reflects_def)
apply (elim meta_exE)
apply (rule_tac x="λa. Cl(a) ∧ Cla(a)" in meta_exI)
apply (simp add: Closed_Unbounded_Int, blast)
done

theorem Or_reflection:
     "[| REFLECTS[P,Q]; REFLECTS[P',Q'] |]
      ==> REFLECTS[λx. P(x) ∨ P'(x), λa x. Q(a,x) ∨ Q'(a,x)]"
apply (unfold L_Reflects_def)
apply (elim meta_exE)
apply (rule_tac x="λa. Cl(a) ∧ Cla(a)" in meta_exI)
apply (simp add: Closed_Unbounded_Int, blast)
done

theorem Imp_reflection:
     "[| REFLECTS[P,Q]; REFLECTS[P',Q'] |]
      ==> REFLECTS[λx. P(x) --> P'(x), λa x. Q(a,x) --> Q'(a,x)]"
apply (unfold L_Reflects_def)
apply (elim meta_exE)
apply (rule_tac x="λa. Cl(a) ∧ Cla(a)" in meta_exI)
apply (simp add: Closed_Unbounded_Int, blast)
done

theorem Iff_reflection:
     "[| REFLECTS[P,Q]; REFLECTS[P',Q'] |]
      ==> REFLECTS[λx. P(x) <-> P'(x), λa x. Q(a,x) <-> Q'(a,x)]"
apply (unfold L_Reflects_def)
apply (elim meta_exE)
apply (rule_tac x="λa. Cl(a) ∧ Cla(a)" in meta_exI)
apply (simp add: Closed_Unbounded_Int, blast)
done


lemma reflection_Lset: "reflection(Lset)"
by (blast intro: reflection.intro Lset_mono_le Lset_cont
                 Formula.Pair_in_LLimit)+


theorem Ex_reflection:
     "REFLECTS[λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x))]
      ==> REFLECTS[λx. ∃z. L(z) ∧ P(x,z), λa x. ∃z∈Lset(a). Q(a,x,z)]"
apply (unfold L_Reflects_def L_ClEx_def L_FF_def L_F0_def L_def)
apply (elim meta_exE)
apply (rule meta_exI)
apply (erule reflection.Ex_reflection [OF reflection_Lset])
done
```

```
theorem All_reflection:
     "REFLECTS[λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x))]
      ==> REFLECTS[λx. ∀z. L(z) --> P(x,z), λa x. ∀z∈Lset(a). Q(a,x,z)]"
apply (unfold L_Reflects_def L_ClEx_def L_FF_def L_F0_def L_def)
apply (elim meta_exE)
apply (rule meta_exI)
apply (erule reflection.All_reflection [OF reflection_Lset])
done

theorem Rex_reflection:
     "REFLECTS[ λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x))]
      ==> REFLECTS[λx. ∃z[L]. P(x,z), λa x. ∃z∈Lset(a). Q(a,x,z)]"
apply (unfold rex_def)
apply (intro And_reflection Ex_reflection, assumption)
done

theorem Rall_reflection:
     "REFLECTS[λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x))]
      ==> REFLECTS[λx. ∀z[L]. P(x,z), λa x. ∀z∈Lset(a). Q(a,x,z)]"
apply (unfold rall_def)
apply (intro Imp_reflection All_reflection, assumption)
done
```

This version handles an alternative form of the bounded quantifier in the second argument of `REFLECTS`.

```
theorem Rex_reflection':
     "REFLECTS[λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x))]
      ==> REFLECTS[λx. ∃z[L]. P(x,z), λa x. ∃z[##Lset(a)]. Q(a,x,z)]"
apply (unfold setclass_def rex_def)
apply (erule Rex_reflection [unfolded rex_def Bex_def])
done
```

As above.

```
theorem Rall_reflection':
     "REFLECTS[λx. P(fst(x),snd(x)), λa x. Q(a,fst(x),snd(x))]
      ==> REFLECTS[λx. ∀z[L]. P(x,z), λa x. ∀z[##Lset(a)]. Q(a,x,z)]"
apply (unfold setclass_def rall_def)
apply (erule Rall_reflection [unfolded rall_def Ball_def])
done

lemmas FOL_reflections =
        Triv_reflection Not_reflection And_reflection Or_reflection
        Imp_reflection Iff_reflection Ex_reflection All_reflection
        Rex_reflection Rall_reflection Rex_reflection' Rall_reflection'

lemma ReflectsD:
     "[|REFLECTS[P,Q]; Ord(i)|]
      ==> ∃j. i<j & (∀x ∈ Lset(j). P(x) <-> Q(j,x))"
apply (unfold L_Reflects_def Closed_Unbounded_def)
```

```
apply (elim meta_exE, clarify)
apply (blast dest!: UnboundedD)
done

lemma ReflectsE:
     "[| REFLECTS[P,Q]; Ord(i);
         !!j. [|i<j;  ∀x ∈ Lset(j). P(x) <-> Q(j,x)|] ==> R |]
      ==> R"
by (drule ReflectsD, assumption, blast)

lemma Collect_mem_eq: "{x∈A. x∈B} = A ∩ B"
by blast
```

10.4 Internalized Formulas for some Set-Theoretic Concepts

10.4.1 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)"
```

10.4.2 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

theorem empty_reflection:
     "REFLECTS[λx. empty(L,f(x)),
               λi x. empty(##Lset(i),f(x))]"
apply (simp only: empty_def)
apply (intro FOL_reflections)
done
```

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
```

10.4.3 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 add: sats_upair_fm)
```

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

theorem upair_reflection:
     "REFLECTS[λx. upair(L,f(x),g(x),h(x)),
               λi x. upair(##Lset(i),f(x),g(x),h(x))]"
apply (simp add: upair_def)
apply (intro FOL_reflections)
done
```

10.4.4 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 add: sats_pair_fm)

theorem pair_reflection:
     "REFLECTS[λx. pair(L,f(x),g(x),h(x)),
               λi x. pair(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: pair_def)
```

```
apply (intro FOL_reflections upair_reflection)
done
```

10.4.5 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 add: sats_union_fm)

theorem union_reflection:
     "REFLECTS[λx. union(L,f(x),g(x),h(x)),
               λi x. union(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: union_def)
apply (intro FOL_reflections)
done
```

10.4.6 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

theorem cons_reflection:
     "REFLECTS[λx. is_cons(L,f(x),g(x),h(x)),
               λi x. is_cons(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_cons_def)
apply (intro FOL_reflections upair_reflection union_reflection)
done
```

10.4.7 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

theorem successor_reflection:
     "REFLECTS[λx. successor(L,f(x),g(x)),
               λi x. successor(##Lset(i),f(x),g(x))]"
apply (simp only: successor_def)
apply (intro cons_reflection)
done
```

10.4.8 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

theorem number1_reflection:
     "REFLECTS[λx. number1(L,f(x)),
               λi x. number1(##Lset(i),f(x))]"
apply (simp only: number1_def)
apply (intro FOL_reflections empty_reflection successor_reflection)
done
```

10.4.9 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

theorem big_union_reflection:
     "REFLECTS[λx. big_union(L,f(x),g(x)),
```

```
              λi x. big_union(##Lset(i),f(x),g(x))]"
apply (simp only: big_union_def)
apply (intro FOL_reflections)
done
```

10.4.10 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)

theorem subset_reflection:
     "REFLECTS[λx. subset(L,f(x),g(x)),
              λi x. subset(##Lset(i),f(x),g(x))]"
apply (simp only: Relative.subset_def)
apply (intro FOL_reflections)
done

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)

theorem transitive_set_reflection:
     "REFLECTS[λx. transitive_set(L,f(x)),
              λi x. transitive_set(##Lset(i),f(x))]"
apply (simp only: transitive_set_def)
apply (intro FOL_reflections subset_reflection)
done

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')

theorem ordinal_reflection:
     "REFLECTS[λx. ordinal(L,f(x)), λi x. ordinal(##Lset(i),f(x))]"
```

apply *(simp only: ordinal_def)*
apply *(intro FOL_reflections transitive_set_reflection)*
done

10.4.11 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*

theorem *membership_reflection:*
```
     "REFLECTS[λx. membership(L,f(x),g(x)),
               λi x. membership(##Lset(i),f(x),g(x))]"
```
apply *(simp only: membership_def)*
apply *(intro FOL_reflections pair_reflection)*
done

10.4.12 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 add: sats_pred_set_fm)

theorem pred_set_reflection:
     "REFLECTS[λx. pred_set(L,f(x),g(x),h(x),b(x)),
               λi x. pred_set(##Lset(i),f(x),g(x),h(x),b(x))]"
apply (simp only: pred_set_def)
apply (intro FOL_reflections pair_reflection)
done
```

10.4.13 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
```

theorem *domain_reflection:*
```
     "REFLECTS[λx. is_domain(L,f(x),g(x)),
               λi x. is_domain(##Lset(i),f(x),g(x))]"
```
apply *(simp only: is_domain_def)*
apply *(intro FOL_reflections pair_reflection)*
done

10.4.14 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*

theorem *range_reflection:*
```
     "REFLECTS[λx. is_range(L,f(x),g(x)),
               λi x. is_range(##Lset(i),f(x),g(x))]"
```
apply *(simp only: is_range_def)*
apply *(intro FOL_reflections pair_reflection)*
done

10.4.15 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

theorem field_reflection:
     "REFLECTS[λx. is_field(L,f(x),g(x)),
               λi x. is_field(##Lset(i),f(x),g(x))]"
apply (simp only: is_field_def)
apply (intro FOL_reflections domain_reflection range_reflection
             union_reflection)
done
```

10.4.16 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 add: sats_image_fm)

theorem image_reflection:
```

```
    "REFLECTS[λx. image(L,f(x),g(x),h(x)),
              λi x. image(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: Relative.image_def)
apply (intro FOL_reflections pair_reflection)
done
```

10.4.17 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 add: sats_pre_image_fm)

theorem pre_image_reflection:
     "REFLECTS[λx. pre_image(L,f(x),g(x),h(x)),
              λi x. pre_image(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: Relative.pre_image_def)
apply (intro FOL_reflections pair_reflection)
done
```

10.4.18 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*

theorem *fun_apply_reflection:*
```
     "REFLECTS[λx. fun_apply(L,f(x),g(x),h(x)),
               λi x. fun_apply(##Lset(i),f(x),g(x),h(x))]"
```
apply *(simp only: fun_apply_def)*
apply *(intro FOL_reflections upair_reflection image_reflection big_union_reflection)*
done

10.4.19 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*

theorem *is_relation_reflection:*
```
     "REFLECTS[λx. is_relation(L,f(x)),
               λi x. is_relation(##Lset(i),f(x))]"
```
apply *(simp only: is_relation_def)*
apply *(intro FOL_reflections pair_reflection)*

done

10.4.20 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

theorem is_function_reflection:
     "REFLECTS[λx. is_function(L,f(x)),
               λi x. is_function(##Lset(i),f(x))]"
apply (simp only: is_function_def)
apply (intro FOL_reflections pair_reflection)
done
```

10.4.21 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_reflections =
        empty_reflection number1_reflection
        upair_reflection pair_reflection union_reflection
        big_union_reflection cons_reflection successor_reflection
        fun_apply_reflection subset_reflection
        transitive_set_reflection membership_reflection
        pred_set_reflection domain_reflection range_reflection field_reflection
        image_reflection pre_image_reflection
        is_relation_reflection is_function_reflection

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


theorem typed_function_reflection:
     "REFLECTS[λx. typed_function(L,f(x),g(x),h(x)),
               λi x. typed_function(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: typed_function_def)
apply (intro FOL_reflections function_reflections)
done
```

10.4.22 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*

theorem *composition_reflection:*
```
     "REFLECTS[λx. composition(L,f(x),g(x),h(x)),
               λi x. composition(##Lset(i),f(x),g(x),h(x))]"
```
apply *(simp only: composition_def)*
apply *(intro FOL_reflections pair_reflection)*
done

10.4.23 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

theorem injection_reflection:
     "REFLECTS[λx. injection(L,f(x),g(x),h(x)),
               λi x. injection(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: injection_def)
apply (intro FOL_reflections function_reflections typed_function_reflection)
done
```

10.4.24 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

theorem surjection_reflection:
     "REFLECTS[λx. surjection(L,f(x),g(x),h(x)),
               λi x. surjection(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: surjection_def)
apply (intro FOL_reflections function_reflections typed_function_reflection)
done
```

10.4.25 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

theorem bijection_reflection:
     "REFLECTS[λx. bijection(L,f(x),g(x),h(x)),
               λi x. bijection(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: bijection_def)
apply (intro And_reflection injection_reflection surjection_reflection)
done
```

10.4.26 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
```

```
theorem restriction_reflection:
     "REFLECTS[λx. restriction(L,f(x),g(x),h(x)),
               λi x. restriction(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: restriction_def)
apply (intro FOL_reflections pair_reflection)
done
```

10.4.27 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
```

```
theorem order_isomorphism_reflection:
```

```
     "REFLECTS[λx. order_isomorphism(L,f(x),g(x),h(x),g'(x),h'(x)),
               λi x. order_isomorphism(##Lset(i),f(x),g(x),h(x),g'(x),h'(x))]"
apply (simp only: order_isomorphism_def)
apply (intro FOL_reflections function_reflections bijection_reflection)
done
```

10.4.28 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

theorem limit_ordinal_reflection:
     "REFLECTS[λx. limit_ordinal(L,f(x)),
               λi x. limit_ordinal(##Lset(i),f(x))]"
apply (simp only: limit_ordinal_def)
apply (intro FOL_reflections ordinal_reflection
             empty_reflection successor_reflection)
done
```

10.4.29 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

theorem finite_ordinal_reflection:
     "REFLECTS[λx. finite_ordinal(L,f(x)),
               λi x. finite_ordinal(##Lset(i),f(x))]"
apply (simp only: finite_ordinal_def)
apply (intro FOL_reflections ordinal_reflection limit_ordinal_reflection)
done
```

10.4.30 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

theorem omega_reflection:
```

```
     "REFLECTS[λx. omega(L,f(x)),
              λi x. omega(##Lset(i),f(x))]"
apply (simp only: omega_def)
apply (intro FOL_reflections limit_ordinal_reflection)
done


lemmas fun_plus_reflections =
        typed_function_reflection composition_reflection
        injection_reflection surjection_reflection
        bijection_reflection restriction_reflection
        order_isomorphism_reflection finite_ordinal_reflection
        ordinal_reflection limit_ordinal_reflection omega_reflection

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

end
```

11 Early Instances of Separation and Strong Replacement

theory *Separation* **imports** *L_axioms WF_absolute* **begin**

This theory proves all instances needed for locale *M_basic*

Helps us solve for de Bruijn indices!

```
lemma nth_ConsI: "[|nth(n,l) = x; n ∈ nat|] ==> nth(succ(n), Cons(a,l))
= x"
by simp

lemmas nth_rules = nth_0 nth_ConsI nat_0I nat_succI
lemmas sep_rules = nth_0 nth_ConsI FOL_iff_sats function_iff_sats
                   fun_plus_iff_sats

lemma Collect_conj_in_DPow:
     "[| {x∈A. P(x)} ∈ DPow(A);  {x∈A. Q(x)} ∈ DPow(A) |]
      ==> {x∈A. P(x) & Q(x)} ∈ DPow(A)"
by (simp add: Int_in_DPow Collect_Int_Collect_eq [symmetric])

lemma Collect_conj_in_DPow_Lset:
     "[|z ∈ Lset(j); {x ∈ Lset(j). P(x)} ∈ DPow(Lset(j))|]
      ==> {x ∈ Lset(j). x ∈ z & P(x)} ∈ DPow(Lset(j))"
```

```
apply (frule mem_Lset_imp_subset_Lset)
apply (simp add: Collect_conj_in_DPow Collect_mem_eq
                 subset_Int_iff2 elem_subset_in_DPow)
done
```

```
lemma separation_CollectI:
     "(⋀z. L(z) ==> L({x ∈ z . P(x)})) ==> separation(L, λx. P(x))"
apply (unfold separation_def, clarify)
apply (rule_tac x="{x∈z. P(x)}" in rexI)
apply simp_all
done
```

Reduces the original comprehension to the reflected one

```
lemma reflection_imp_L_separation:
      "[| ∀x∈Lset(j). P(x) <-> Q(x);
          {x ∈ Lset(j) . Q(x)} ∈ DPow(Lset(j));
          Ord(j);  z ∈ Lset(j)|] ==> L({x ∈ z . P(x)})"
apply (rule_tac i = "succ(j)" in L_I)
 prefer 2 apply simp
apply (subgoal_tac "{x ∈ z. P(x)} = {x ∈ Lset(j). x ∈ z & (Q(x))}")
 prefer 2
 apply (blast dest: mem_Lset_imp_subset_Lset)
apply (simp add: Lset_succ Collect_conj_in_DPow_Lset)
done
```

Encapsulates the standard proof script for proving instances of Separation.

```
lemma gen_separation:
 assumes reflection: "REFLECTS [P,Q]"
     and Lu:         "L(u)"
     and collI: "!!j. u ∈ Lset(j)
                ⟹ Collect(Lset(j), Q(j)) ∈ DPow(Lset(j))"
 shows "separation(L,P)"
apply (rule separation_CollectI)
apply (rule_tac A="{u,z}" in subset_LsetE, blast intro: Lu)
apply (rule ReflectsE [OF reflection], assumption)
apply (drule subset_Lset_ltD, assumption)
apply (erule reflection_imp_L_separation)
  apply (simp_all add: lt_Ord2, clarify)
apply (rule collI, assumption)
done
```

As above, but typically `u` is a finite enumeration such as `{a, b}`; thus the new subgoal gets the assumption `{a, b} ⊆ Lset(i)`, which is logically equivalent to `a ∈ Lset(i)` and `b ∈ Lset(i)`.

```
lemma gen_separation_multi:
 assumes reflection: "REFLECTS [P,Q]"
     and Lu:         "L(u)"
     and collI: "!!j. u ⊆ Lset(j)
```

```
              ⟹ Collect(Lset(j), Q(j)) ∈ DPow(Lset(j))"
 shows "separation(L,P)"
apply (rule gen_separation [OF reflection Lu])
apply (drule mem_Lset_imp_subset_Lset)
apply (erule collI)
done
```

11.1 Separation for Intersection

```
lemma Inter_Reflects:
     "REFLECTS[λx. ∀y[L]. y∈A --> x ∈ y,
               λi x. ∀y∈Lset(i). y∈A --> x ∈ y]"
by (intro FOL_reflections)

lemma Inter_separation:
     "L(A) ==> separation(L, λx. ∀y[L]. y∈A --> x∈y)"
apply (rule gen_separation [OF Inter_Reflects], simp)
apply (rule DPow_LsetI)
```

I leave this one example of a manual proof. The tedium of manually instantiating *i*, *j* and *env* is obvious.

```
apply (rule ball_iff_sats)
apply (rule imp_iff_sats)
apply (rule_tac [2] i=1 and j=0 and env="[y,x,A]" in mem_iff_sats)
apply (rule_tac i=0 and j=2 in mem_iff_sats)
apply (simp_all add: succ_Un_distrib [symmetric])
done
```

11.2 Separation for Set Difference

```
lemma Diff_Reflects:
     "REFLECTS[λx. x ∉ B, λi x. x ∉ B]"
by (intro FOL_reflections)

lemma Diff_separation:
     "L(B) ==> separation(L, λx. x ∉ B)"
apply (rule gen_separation [OF Diff_Reflects], simp)
apply (rule_tac env="[B]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

11.3 Separation for Cartesian Product

```
lemma cartprod_Reflects:
     "REFLECTS[λz. ∃x[L]. x∈A & (∃y[L]. y∈B & pair(L,x,y,z)),
               λi z. ∃x∈Lset(i). x∈A & (∃y∈Lset(i). y∈B &
                                    pair(##Lset(i),x,y,z))]"
by (intro FOL_reflections function_reflections)

lemma cartprod_separation:
```

```
     "[| L(A); L(B) |]
      ==> separation(L, λz. ∃x[L]. x∈A & (∃y[L]. y∈B & pair(L,x,y,z)))"
apply (rule gen_separation_multi [OF cartprod_Reflects, of "{A,B}"], auto)
apply (rule_tac env="[A,B]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

11.4 Separation for Image

```
lemma image_Reflects:
     "REFLECTS[λy. ∃p[L]. p∈r & (∃x[L]. x∈A & pair(L,x,y,p)),
           λi y. ∃p∈Lset(i). p∈r & (∃x∈Lset(i). x∈A & pair(##Lset(i),x,y,p))]"
by (intro FOL_reflections function_reflections)

lemma image_separation:
     "[| L(A); L(r) |]
      ==> separation(L, λy. ∃p[L]. p∈r & (∃x[L]. x∈A & pair(L,x,y,p)))"
apply (rule gen_separation_multi [OF image_Reflects, of "{A,r}"], auto)
apply (rule_tac env="[A,r]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

11.5 Separation for Converse

```
lemma converse_Reflects:
  "REFLECTS[λz. ∃p[L]. p∈r & (∃x[L]. ∃y[L]. pair(L,x,y,p) & pair(L,y,x,z)),
     λi z. ∃p∈Lset(i). p∈r & (∃x∈Lset(i). ∃y∈Lset(i).
                    pair(##Lset(i),x,y,p) & pair(##Lset(i),y,x,z))]"
by (intro FOL_reflections function_reflections)

lemma converse_separation:
     "L(r) ==> separation(L,
         λz. ∃p[L]. p∈r & (∃x[L]. ∃y[L]. pair(L,x,y,p) & pair(L,y,x,z)))"
apply (rule gen_separation [OF converse_Reflects], simp)
apply (rule_tac env="[r]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

11.6 Separation for Restriction

```
lemma restrict_Reflects:
     "REFLECTS[λz. ∃x[L]. x∈A & (∃y[L]. pair(L,x,y,z)),
        λi z. ∃x∈Lset(i). x∈A & (∃y∈Lset(i). pair(##Lset(i),x,y,z))]"
by (intro FOL_reflections function_reflections)

lemma restrict_separation:
   "L(A) ==> separation(L, λz. ∃x[L]. x∈A & (∃y[L]. pair(L,x,y,z)))"
apply (rule gen_separation [OF restrict_Reflects], simp)
apply (rule_tac env="[A]" in DPow_LsetI)
apply (rule sep_rules | simp)+
```

done

11.7 Separation for Composition

lemma *comp_Reflects:*
```
"REFLECTS[λxz. ∃x[L]. ∃y[L]. ∃z[L]. ∃xy[L]. ∃yz[L].
               pair(L,x,z,xz) & pair(L,x,y,xy) & pair(L,y,z,yz) &
               xy∈s & yz∈r,
    λi xz. ∃x∈Lset(i). ∃y∈Lset(i). ∃z∈Lset(i). ∃xy∈Lset(i). ∃yz∈Lset(i).
               pair(##Lset(i),x,z,xz) & pair(##Lset(i),x,y,xy) &
               pair(##Lset(i),y,z,yz) & xy∈s & yz∈r]"
```
by *(intro FOL_reflections function_reflections)*

lemma *comp_separation:*
```
"[| L(r); L(s) |]
 ==> separation(L, λxz. ∃x[L]. ∃y[L]. ∃z[L]. ∃xy[L]. ∃yz[L].
               pair(L,x,z,xz) & pair(L,x,y,xy) & pair(L,y,z,yz) &
               xy∈s & yz∈r)"
```
apply *(rule gen_separation_multi [OF comp_Reflects, of "{r,s}"], auto)*

Subgoals after applying general "separation" rule:

```
1. ⋀j. ⟦L(r); L(s); r ∈ Lset(j); s ∈ Lset(j)⟧
       ⟹ {xz ∈ Lset(j) .
            ∃x∈Lset(j).
               ∃y∈Lset(j).
                  ∃z∈Lset(j).
                     pair(##Lset(j), x, z, xz) ∧
                     (∃xy∈Lset(j).
                         pair(##Lset(j), x, y, xy) ∧
                         (∃yz∈Lset(j).
                             pair(##Lset(j), y, z, yz) ∧
                             xy ∈ s ∧ yz ∈ r))} ∈
          DPow(Lset(j))
```

apply *(rule_tac env="[r,s]"* **in** *DPow_LsetI)*

Subgoals ready for automatic synthesis of a formula:

```
1. ⋀j x. ⟦L(r); L(s); r ∈ Lset(j); s ∈ Lset(j); x ∈ Lset(j)⟧
         ⟹ (∃xa∈Lset(j).
                 ∃y∈Lset(j).
                    ∃z∈Lset(j).
                       pair(##Lset(j), xa, z, x) ∧
                       (∃xy∈Lset(j).
                           pair(##Lset(j), xa, y, xy) ∧
                           (∃yz∈Lset(j).
                               pair(##Lset(j), y, z, yz) ∧
                               xy ∈ s ∧ yz ∈ r))) ⟷
             sats(Lset(j), ?p18(j), [x, r, s])
```

```
 2. ⋀j. ⟦L(r); L(s); r ∈ Lset(j); s ∈ Lset(j)⟧
        ⟹ [r, s] ∈ list(Lset(j))
 3. ⋀j. ⟦L(r); L(s); r ∈ Lset(j); s ∈ Lset(j)⟧
        ⟹ ?p18(j) ∈ formula
```

apply *(rule sep_rules | simp)+*
done

11.8 Separation for Predecessors in an Order

```
lemma pred_Reflects:
     "REFLECTS[λy. ∃p[L]. p∈r & pair(L,y,x,p),
                    λi y. ∃p ∈ Lset(i). p∈r & pair(##Lset(i),y,x,p)]"
by (intro FOL_reflections function_reflections)

lemma pred_separation:
     "[| L(r); L(x) |] ==> separation(L, λy. ∃p[L]. p∈r & pair(L,y,x,p))"
apply (rule gen_separation_multi [OF pred_Reflects, of "{r,x}"], auto)
apply (rule_tac env="[r,x]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

11.9 Separation for the Membership Relation

```
lemma Memrel_Reflects:
     "REFLECTS[λz. ∃x[L]. ∃y[L]. pair(L,x,y,z) & x ∈ y,
            λi z. ∃x ∈ Lset(i). ∃y ∈ Lset(i). pair(##Lset(i),x,y,z)
& x ∈ y]"
by (intro FOL_reflections function_reflections)

lemma Memrel_separation:
     "separation(L, λz. ∃x[L]. ∃y[L]. pair(L,x,y,z) & x ∈ y)"
apply (rule gen_separation [OF Memrel_Reflects nonempty])
apply (rule_tac env="[]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

11.10 Replacement for FunSpace

```
lemma funspace_succ_Reflects:
 "REFLECTS[λz. ∃p[L]. p∈A & (∃f[L]. ∃b[L]. ∃nb[L]. ∃cnbf[L].
            pair(L,f,b,p) & pair(L,n,b,nb) & is_cons(L,nb,f,cnbf) &
            upair(L,cnbf,cnbf,z)),
        λi z. ∃p ∈ Lset(i). p∈A & (∃f ∈ Lset(i). ∃b ∈ Lset(i).
              ∃nb ∈ Lset(i). ∃cnbf ∈ Lset(i).
                pair(##Lset(i),f,b,p) & pair(##Lset(i),n,b,nb) &
                is_cons(##Lset(i),nb,f,cnbf) & upair(##Lset(i),cnbf,cnbf,z))]"
by (intro FOL_reflections function_reflections)

lemma funspace_succ_replacement:
```

```
     "L(n) ==>
      strong_replacement(L, λp z. ∃f[L]. ∃b[L]. ∃nb[L]. ∃cnbf[L].
                pair(L,f,b,p) & pair(L,n,b,nb) & is_cons(L,nb,f,cnbf)
&
                upair(L,cnbf,cnbf,z))"
apply (rule strong_replacementI)
apply (rule_tac u="{n,B}" in gen_separation_multi [OF funspace_succ_Reflects],

       auto)
apply (rule_tac env="[n,B]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

11.11 Separation for a Theorem about *is_recfun*

```
lemma is_recfun_reflects:
  "REFLECTS[λx. ∃xa[L]. ∃xb[L].
                pair(L,x,a,xa) & xa ∈ r & pair(L,x,b,xb) & xb ∈ r &
                (∃fx[L]. ∃gx[L]. fun_apply(L,f,x,fx) & fun_apply(L,g,x,gx)
&
                                   fx ≠ gx),
   λi x. ∃xa ∈ Lset(i). ∃xb ∈ Lset(i).
          pair(##Lset(i),x,a,xa) & xa ∈ r & pair(##Lset(i),x,b,xb) & xb
∈ r &
                (∃fx ∈ Lset(i). ∃gx ∈ Lset(i). fun_apply(##Lset(i),f,x,fx)
&
                  fun_apply(##Lset(i),g,x,gx) & fx ≠ gx)]"
by (intro FOL_reflections function_reflections fun_plus_reflections)

lemma is_recfun_separation:
     — for well-founded recursion
     "[| L(r); L(f); L(g); L(a); L(b) |]
     ==> separation(L,
          λx. ∃xa[L]. ∃xb[L].
                pair(L,x,a,xa) & xa ∈ r & pair(L,x,b,xb) & xb ∈ r &
                (∃fx[L]. ∃gx[L]. fun_apply(L,f,x,fx) & fun_apply(L,g,x,gx)
&
                                   fx ≠ gx))"
apply (rule gen_separation_multi [OF is_recfun_reflects, of "{r,f,g,a,b}"],

          auto)
apply (rule_tac env="[r,f,g,a,b]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

11.12 Instantiating the locale *M_basic*

Separation (and Strong Replacement) for basic set-theoretic constructions such as intersection, Cartesian Product and image.

```
lemma M_basic_axioms_L: "M_basic_axioms(L)"
  apply (rule M_basic_axioms.intro)
       apply (assumption | rule
         Inter_separation Diff_separation cartprod_separation image_separation
         converse_separation restrict_separation
         comp_separation pred_separation Memrel_separation
         funspace_succ_replacement is_recfun_separation)+
  done

theorem M_basic_L: "PROP M_basic(L)"
by (rule M_basic.intro [OF M_trivial_L M_basic_axioms_L])

interpretation M_basic [L] by (rule M_basic_L)


end
```

```
theory Internalize imports L_axioms Datatype_absolute begin
```

11.13 Internalized Forms of Data Structuring Operators

11.13.1 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

theorem Inl_reflection:
     "REFLECTS[λx. is_Inl(L,f(x),h(x)),
               λi x. is_Inl(##Lset(i),f(x),h(x))]"
apply (simp only: is_Inl_def)
apply (intro FOL_reflections function_reflections)
done
```

11.13.2 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*

theorem *Inr_reflection:*
```
     "REFLECTS[λx. is_Inr(L,f(x),h(x)),
               λi x. is_Inr(##Lset(i),f(x),h(x))]"
```
apply *(simp only: is_Inr_def)*
apply *(intro FOL_reflections function_reflections)*
done

11.13.3 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*

theorem *Nil_reflection:*
```
     "REFLECTS[λx. is_Nil(L,f(x)),
               λi x. is_Nil(##Lset(i),f(x))]"
```

apply *(simp only: is_Nil_def)*
apply *(intro FOL_reflections function_reflections Inl_reflection)*
done

11.13.4 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

theorem Cons_reflection:
     "REFLECTS[λx. is_Cons(L,f(x),g(x),h(x)),
               λi x. is_Cons(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_Cons_def)
apply (intro FOL_reflections pair_reflection Inr_reflection)
done
```

11.13.5 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

theorem quasilist_reflection:
     "REFLECTS[λx. is_quasilist(L,f(x)),
               λi x. is_quasilist(##Lset(i),f(x))]"
apply (simp only: is_quasilist_def)
apply (intro FOL_reflections Nil_reflection Cons_reflection)
done
```

11.14 Absoluteness for the Function *nth*

11.14.1 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

theorem hd_reflection:
     "REFLECTS[λx. is_hd(L,f(x),g(x)),
               λi x. is_hd(##Lset(i),f(x),g(x))]"
apply (simp only: is_hd_def)
apply (intro FOL_reflections Nil_reflection Cons_reflection
             quasilist_reflection empty_reflection)
done
```

11.14.2 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*

theorem *tl_reflection:*
```
     "REFLECTS[λx. is_tl(L,f(x),g(x)),
               λi x. is_tl(##Lset(i),f(x),g(x))]"
```
apply *(simp only: is_tl_def)*
apply *(intro FOL_reflections Nil_reflection Cons_reflection
            quasilist_reflection empty_reflection)*
done

11.14.3 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)

theorem bool_of_o_reflection:
     "REFLECTS [P(L), λi. P(##Lset(i))] ==>
      REFLECTS[λx. is_bool_of_o(L, P(L,x), f(x)),
               λi x. is_bool_of_o(##Lset(i), P(##Lset(i),x), f(x))]"
apply (simp (no_asm) only: is_bool_of_o_def)
apply (intro FOL_reflections function_reflections, assumption+)
done
```

11.15 More Internalizations

11.15.1 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])


theorem is_lambda_reflection:
  assumes is_b_reflection:
    "!!f g h. REFLECTS[λx. is_b(L, f(x), g(x), h(x)),
                      λi x. is_b(##Lset(i), f(x), g(x), h(x))]"
  shows "REFLECTS[λx. is_lambda(L, A(x), is_b(L,x), f(x)),
```

```
              λi x. is_lambda(##Lset(i), A(x), is_b(##Lset(i),x), f(x))]"
apply (simp (no_asm_use) only: is_lambda_def)
apply (intro FOL_reflections is_b_reflection pair_reflection)
done
```

11.15.2 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)

theorem Member_reflection:
     "REFLECTS[λx. is_Member(L,f(x),g(x),h(x)),
              λi x. is_Member(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_Member_def)
apply (intro FOL_reflections pair_reflection Inl_reflection)
done
```

11.15.3 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)

theorem Equal_reflection:
     "REFLECTS[λx. is_Equal(L,f(x),g(x),h(x)),
               λi x. is_Equal(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_Equal_def)
apply (intro FOL_reflections pair_reflection Inl_reflection Inr_reflection)
done
```

11.15.4 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)

theorem Nand_reflection:
     "REFLECTS[λx. is_Nand(L,f(x),g(x),h(x)),
               λi x. is_Nand(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_Nand_def)
apply (intro FOL_reflections pair_reflection Inl_reflection Inr_reflection)
done
```

11.15.5 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)

theorem Forall_reflection:
     "REFLECTS[λx. is_Forall(L,f(x),g(x)),
               λi x. is_Forall(##Lset(i),f(x),g(x))]"
apply (simp only: is_Forall_def)
apply (intro FOL_reflections pair_reflection Inr_reflection)
done
```

11.15.6 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

theorem is_and_reflection:
     "REFLECTS[λx. is_and(L,f(x),g(x),h(x)),
               λi x. is_and(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_and_def)
apply (intro FOL_reflections function_reflections)
done
```

11.15.7 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

theorem is_or_reflection:
     "REFLECTS[λx. is_or(L,f(x),g(x),h(x)),
               λi x. is_or(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_or_def)
apply (intro FOL_reflections function_reflections)
done
```

11.15.8 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

theorem is_not_reflection:
     "REFLECTS[λx. is_not(L,f(x),g(x)),
               λi x. is_not(##Lset(i),f(x),g(x))]"
apply (simp only: is_not_def)
apply (intro FOL_reflections function_reflections)
done


lemmas extra_reflections =
    Inl_reflection Inr_reflection Nil_reflection Cons_reflection
    quasilist_reflection hd_reflection tl_reflection bool_of_o_reflection
    is_lambda_reflection Member_reflection Equal_reflection Nand_reflection
    Forall_reflection is_and_reflection is_or_reflection is_not_reflection
```

11.16 Well-Founded Recursion!

11.16.1 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*.

```
theorem is_recfun_reflection:
  assumes MH_reflection:
    "!!f' f g h. REFLECTS[λx. MH(L, f'(x), f(x), g(x), h(x)),
                     λi x. MH(##Lset(i), f'(x), f(x), g(x), h(x))]"
  shows "REFLECTS[λx. M_is_recfun(L, MH(L,x), f(x), g(x), h(x)),
```

```
             λi x. M_is_recfun(##Lset(i), MH(##Lset(i),x), f(x), g(x),
h(x))]"
apply (simp (no_asm_use) only: M_is_recfun_def)
apply (intro FOL_reflections function_reflections
             restriction_reflection MH_reflection)
done
```

11.16.2 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])*

```
theorem is_wfrec_reflection:
  assumes MH_reflection:
    "!!f' f g h. REFLECTS[λx. MH(L, f'(x), f(x), g(x), h(x)),
                    λi x. MH(##Lset(i), f'(x), f(x), g(x), h(x))]"
  shows "REFLECTS[λx. is_wfrec(L, MH(L,x), f(x), g(x), h(x)),
               λi x. is_wfrec(##Lset(i), MH(##Lset(i),x), f(x), g(x),
h(x))]"
apply (simp (no_asm_use) only: is_wfrec_def)
apply (intro FOL_reflections MH_reflection is_recfun_reflection)
done
```

11.17 For Datatypes

11.17.1 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)

theorem cartprod_reflection:
     "REFLECTS[λx. cartprod(L,f(x),g(x),h(x)),
               λi x. cartprod(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: cartprod_def)
```

apply *(intro FOL_reflections pair_reflection)*
done

11.17.2 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

theorem sum_reflection:
     "REFLECTS[λx. is_sum(L,f(x),g(x),h(x)),
               λi x. is_sum(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_sum_def)
apply (intro FOL_reflections function_reflections cartprod_reflection)
done
```

11.17.3 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

theorem quasinat_reflection:
     "REFLECTS[λx. is_quasinat(L,f(x)),
               λi x. is_quasinat(##Lset(i),f(x))]"
apply (simp only: is_quasinat_def)
apply (intro FOL_reflections function_reflections)
done
```

11.17.4 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`.

```
theorem is_nat_case_reflection:
  assumes is_b_reflection:
    "!!h f g. REFLECTS[λx. is_b(L, h(x), f(x), g(x)),
                     λi x. is_b(##Lset(i), h(x), f(x), g(x))]"
  shows "REFLECTS[λx. is_nat_case(L, f(x), is_b(L,x), g(x), h(x)),
               λi x. is_nat_case(##Lset(i), f(x), is_b(##Lset(i), x),
g(x), h(x))]"
apply (simp (no_asm_use) only: is_nat_case_def)
apply (intro FOL_reflections function_reflections
             restriction_reflection is_b_reflection quasinat_reflection)
done
```

11.18 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`.

```
theorem iterates_MH_reflection:
  assumes p_reflection:
    "!!f g h. REFLECTS[λx. p(L, h(x), f(x), g(x)),
                       λi x. p(##Lset(i), h(x), f(x), g(x))]"
 shows "REFLECTS[λx. iterates_MH(L, p(L,x), e(x), f(x), g(x), h(x)),
              λi x. iterates_MH(##Lset(i), p(##Lset(i),x), e(x), f(x),
g(x), h(x))]"
apply (simp (no_asm_use) only: iterates_MH_def)
apply (intro FOL_reflections function_reflections is_nat_case_reflection
             restriction_reflection p_reflection)
done
```

11.18.1 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`.

```
theorem is_iterates_reflection:
  assumes p_reflection:
    "!!f g h. REFLECTS[λx. p(L, h(x), f(x), g(x)),
                      λi x. p(##Lset(i), h(x), f(x), g(x))]"
 shows "REFLECTS[λx. is_iterates(L, p(L,x), f(x), g(x), h(x)),
               λi x. is_iterates(##Lset(i), p(##Lset(i),x), f(x), g(x),
h(x))]"
apply (simp (no_asm_use) only: is_iterates_def)
apply (intro FOL_reflections function_reflections p_reflection
             is_wfrec_reflection iterates_MH_reflection)
done
```

11.18.2 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)

theorem eclose_n_reflection:
     "REFLECTS[λx. is_eclose_n(L, f(x), g(x), h(x)),
               λi x. is_eclose_n(##Lset(i), f(x), g(x), h(x))]"
apply (simp only: is_eclose_n_def)
apply (intro FOL_reflections function_reflections is_iterates_reflection)

done
```

11.18.3 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*

theorem *mem_eclose_reflection:*
"REFLECTS[λx. mem_eclose(L,f(x),g(x)),
λi x. mem_eclose(##Lset(i),f(x),g(x))]"
apply *(simp only: mem_eclose_def)*
apply *(intro FOL_reflections finite_ordinal_reflection eclose_n_reflection)*
done

11.18.4 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*

theorem *is_eclose_reflection:*
```
     "REFLECTS[λx. is_eclose(L,f(x),g(x)),
               λi x. is_eclose(##Lset(i),f(x),g(x))]"
```
apply *(simp only: is_eclose_def)*
apply *(intro FOL_reflections mem_eclose_reflection)*
done

11.18.5 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*

theorem *list_functor_reflection:*
```
     "REFLECTS[λx. is_list_functor(L,f(x),g(x),h(x)),
               λi x. is_list_functor(##Lset(i),f(x),g(x),h(x))]"
```
apply *(simp only: is_list_functor_def)*
apply *(intro FOL_reflections number1_reflection*
cartprod_reflection sum_reflection)
done

11.18.6 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)*

theorem *list_N_reflection:*
```
     "REFLECTS[λx. is_list_N(L, f(x), g(x), h(x)),
               λi x. is_list_N(##Lset(i), f(x), g(x), h(x))]"
```
apply *(simp only: is_list_N_def)*
apply *(intro FOL_reflections function_reflections*
is_iterates_reflection list_functor_reflection)
done

11.18.7 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

theorem mem_list_reflection:
     "REFLECTS[λx. mem_list(L,f(x),g(x)),
               λi x. mem_list(##Lset(i),f(x),g(x))]"
apply (simp only: mem_list_def)
apply (intro FOL_reflections finite_ordinal_reflection list_N_reflection)
done
```

11.18.8 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

theorem is_list_reflection:
     "REFLECTS[λx. is_list(L,f(x),g(x)),
               λi x. is_list(##Lset(i),f(x),g(x))]"
apply (simp only: is_list_def)
apply (intro FOL_reflections mem_list_reflection)
done
```

11.18.9 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

theorem formula_functor_reflection:
     "REFLECTS[λx. is_formula_functor(L,f(x),g(x)),
               λi x. is_formula_functor(##Lset(i),f(x),g(x))]"
apply (simp only: is_formula_functor_def)
apply (intro FOL_reflections omega_reflection
             cartprod_reflection sum_reflection)
done
```

11.18.10 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)

theorem formula_N_reflection:
     "REFLECTS[λx. is_formula_N(L, f(x), g(x)),
               λi x. is_formula_N(##Lset(i), f(x), g(x))]"
apply (simp only: is_formula_N_def)
apply (intro FOL_reflections function_reflections
             is_iterates_reflection formula_functor_reflection)
done
```

11.18.11 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

theorem mem_formula_reflection:
     "REFLECTS[λx. mem_formula(L,f(x)),
               λi x. mem_formula(##Lset(i),f(x))]"
apply (simp only: mem_formula_def)
apply (intro FOL_reflections finite_ordinal_reflection formula_N_reflection)
done
```

11.18.12 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 ∈ nat ==> is_formula_fm(x) ∈ formula"
by (simp add: is_formula_fm_def)

lemma sats_is_formula_fm [simp]:
   "[| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, is_formula_fm(x), env) <-> 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 ∈ nat; env ∈ list(A)|]
       ==> is_formula(##A, x) <-> sats(A, is_formula_fm(i), env)"
by simp

theorem is_formula_reflection:
     "REFLECTS[λx. is_formula(L,f(x)),
               λi x. is_formula(##Lset(i),f(x))]"
apply (simp only: is_formula_def)
apply (intro FOL_reflections mem_formula_reflection)
done
```

11.18.13 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 ∈ formula; x ∈ nat; z ∈ nat |]
      ==> is_transrec_fm(p,x,z) ∈ 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∈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
      "[|x < length(env); z < length(env); env ∈ list(A)|]
       ==> sats(A, is_transrec_fm(p,x,z), env) <->
           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])

theorem is_transrec_reflection:
  assumes MH_reflection:
    "!!f' f g h. REFLECTS[λx. MH(L, f'(x), f(x), g(x), h(x)),
                    λi x. MH(##Lset(i), f'(x), f(x), g(x), h(x))]"
  shows "REFLECTS[λx. is_transrec(L, MH(L,x), f(x), h(x)),
                λi x. is_transrec(##Lset(i), MH(##Lset(i),x), f(x), h(x))]"
apply (simp (no_asm_use) only: is_transrec_def)
apply (intro FOL_reflections function_reflections MH_reflection
             is_wfrec_reflection is_eclose_reflection)
done

end
```

12 Separation for Facts About Recursion

theory *Rec_Separation* **imports** *Separation Internalize* **begin**

This theory proves all instances needed for locales `M_trancl` and `M_datatypes`

```
lemma eq_succ_imp_lt: "[|i = succ(j); Ord(i)|] ==> j<i"
by simp
```

12.1 The Locale *M_trancl*

12.1.1 Separation for Reflexive/Transitive Closure

First, The Defining Formula

```
definition
  rtran_closure_mem_fm :: "[i,i,i]=>i" where
 "rtran_closure_mem_fm(A,r,p) ==
   Exists(Exists(Exists(
    And(omega_fm(2),
     And(Member(1,2),
      And(succ_fm(1,0),
       Exists(And(typed_function_fm(1, A#+4, 0),
        And(Exists(Exists(Exists(
              And(pair_fm(2,1,p#+7),
               And(empty_fm(0),
                And(fun_apply_fm(3,0,2), fun_apply_fm(3,5,1))))))),
            Forall(Implies(Member(0,3),
             Exists(Exists(Exists(Exists(
              And(fun_apply_fm(5,4,3),
               And(succ_fm(4,2),
                And(fun_apply_fm(5,2,1),
                 And(pair_fm(3,1,0), Member(0,r#+9))))))))))))))))))))"


lemma rtran_closure_mem_type [TC]:
 "[| x ∈ nat; y ∈ nat; z ∈ nat |] ==> rtran_closure_mem_fm(x,y,z) ∈
formula"
by (simp add: rtran_closure_mem_fm_def)

lemma sats_rtran_closure_mem_fm [simp]:
   "[| x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, rtran_closure_mem_fm(x,y,z), env) <->
        rtran_closure_mem(##A, nth(x,env), nth(y,env), nth(z,env))"
by (simp add: rtran_closure_mem_fm_def rtran_closure_mem_def)

lemma rtran_closure_mem_iff_sats:
      "[| nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
          i ∈ nat; j ∈ nat; k ∈ nat; env ∈ list(A)|]
       ==> rtran_closure_mem(##A, x, y, z) <-> sats(A, rtran_closure_mem_fm(i,j,k),
env)"
by (simp add: sats_rtran_closure_mem_fm)

lemma rtran_closure_mem_reflection:
     "REFLECTS[λx. rtran_closure_mem(L,f(x),g(x),h(x)),
               λi x. rtran_closure_mem(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: rtran_closure_mem_def)
apply (intro FOL_reflections function_reflections fun_plus_reflections)
done
```

Separation for `r^*`.

```
lemma rtrancl_separation:
     "[| L(r); L(A) |] ==> separation (L, rtran_closure_mem(L,A,r))"
apply (rule gen_separation_multi [OF rtran_closure_mem_reflection, of
"{r,A}"],
       auto)
apply (rule_tac env="[r,A]" in DPow_LsetI)
apply (rule rtran_closure_mem_iff_sats sep_rules | simp)+
done
```

12.1.2 Reflexive/Transitive Closure, Internalized

```
definition
  rtran_closure_fm :: "[i,i]=>i" where
  "rtran_closure_fm(r,s) ==
   Forall(Implies(field_fm(succ(r),0),
                  Forall(Iff(Member(0,succ(succ(s))),
                             rtran_closure_mem_fm(1,succ(succ(r)),0)))))"

lemma rtran_closure_type [TC]:
     "[| x ∈ nat; y ∈ nat |] ==> rtran_closure_fm(x,y) ∈ formula"
by (simp add: rtran_closure_fm_def)

lemma sats_rtran_closure_fm [simp]:
   "[| x ∈ nat; y ∈ nat; env ∈ list(A)|]
    ==> sats(A, rtran_closure_fm(x,y), env) <->
        rtran_closure(##A, nth(x,env), nth(y,env))"
by (simp add: rtran_closure_fm_def rtran_closure_def)

lemma rtran_closure_iff_sats:
      "[| nth(i,env) = x; nth(j,env) = y;
          i ∈ nat; j ∈ nat; env ∈ list(A)|]
       ==> rtran_closure(##A, x, y) <-> sats(A, rtran_closure_fm(i,j),
env)"
by simp

theorem rtran_closure_reflection:
     "REFLECTS[λx. rtran_closure(L,f(x),g(x)),
               λi x. rtran_closure(##Lset(i),f(x),g(x))]"
apply (simp only: rtran_closure_def)
apply (intro FOL_reflections function_reflections rtran_closure_mem_reflection)
done
```

12.1.3 Transitive Closure of a Relation, Internalized

```
definition
  tran_closure_fm :: "[i,i]=>i" where
  "tran_closure_fm(r,s) ==
   Exists(And(rtran_closure_fm(succ(r),0), composition_fm(succ(r),0,succ(s))))"
```

```
lemma tran_closure_type [TC]:
     "[| x ∈ nat; y ∈ nat |] ==> tran_closure_fm(x,y) ∈ formula"
by (simp add: tran_closure_fm_def)

lemma sats_tran_closure_fm [simp]:
   "[| x ∈ nat; y ∈ nat; env ∈ list(A)|]
    ==> sats(A, tran_closure_fm(x,y), env) <->
        tran_closure(##A, nth(x,env), nth(y,env))"
by (simp add: tran_closure_fm_def tran_closure_def)

lemma tran_closure_iff_sats:
      "[| nth(i,env) = x; nth(j,env) = y;
          i ∈ nat; j ∈ nat; env ∈ list(A)|]
       ==> tran_closure(##A, x, y) <-> sats(A, tran_closure_fm(i,j), env)"
by simp

theorem tran_closure_reflection:
     "REFLECTS[λx. tran_closure(L,f(x),g(x)),
               λi x. tran_closure(##Lset(i),f(x),g(x))]"
apply (simp only: tran_closure_def)
apply (intro FOL_reflections function_reflections
             rtran_closure_reflection composition_reflection)
done
```

12.1.4 Separation for the Proof of `wellfounded_on_trancl`

```
lemma wellfounded_trancl_reflects:
  "REFLECTS[λx. ∃w[L]. ∃wx[L]. ∃rp[L].
                w ∈ Z & pair(L,w,x,wx) & tran_closure(L,r,rp) & wx ∈
rp,
   λi x. ∃w ∈ Lset(i). ∃wx ∈ Lset(i). ∃rp ∈ Lset(i).
       w ∈ Z & pair(##Lset(i),w,x,wx) & tran_closure(##Lset(i),r,rp) &
       wx ∈ rp]"
by (intro FOL_reflections function_reflections fun_plus_reflections
          tran_closure_reflection)

lemma wellfounded_trancl_separation:
         "[| L(r); L(Z) |] ==>
          separation (L, λx.
              ∃w[L]. ∃wx[L]. ∃rp[L].
               w ∈ Z & pair(L,w,x,wx) & tran_closure(L,r,rp) & wx ∈ rp)"
apply (rule gen_separation_multi [OF wellfounded_trancl_reflects, of "{r,Z}"],
       auto)
apply (rule_tac env="[r,Z]" in DPow_LsetI)
apply (rule sep_rules tran_closure_iff_sats | simp)+
done
```

12.1.5 Instantiating the locale *M_trancl*

```
lemma M_trancl_axioms_L: "M_trancl_axioms(L)"
  apply (rule M_trancl_axioms.intro)
   apply (assumption | rule rtrancl_separation wellfounded_trancl_separation)+
  done

theorem M_trancl_L: "PROP M_trancl(L)"
by (rule M_trancl.intro [OF M_basic_L M_trancl_axioms_L])

interpretation M_trancl [L] by (rule M_trancl_L)
```

12.2 *L* is Closed Under the Operator *list*

12.2.1 Instances of Replacement for Lists

```
lemma list_replacement1_Reflects:
 "REFLECTS
   [λx. ∃u[L]. u ∈ B ∧ (∃y[L]. pair(L,u,y,x) ∧
         is_wfrec(L, iterates_MH(L, is_list_functor(L,A), 0), memsn, u,
y)),
    λi x. ∃u ∈ Lset(i). u ∈ B ∧ (∃y ∈ Lset(i). pair(##Lset(i), u, y,
x) ∧
         is_wfrec(##Lset(i),
                  iterates_MH(##Lset(i),
                          is_list_functor(##Lset(i), A), 0), memsn, u,
y))]"
by (intro FOL_reflections function_reflections is_wfrec_reflection
          iterates_MH_reflection list_functor_reflection)


lemma list_replacement1:
   "L(A) ==> iterates_replacement(L, is_list_functor(L,A), 0)"
apply (unfold iterates_replacement_def wfrec_replacement_def, clarify)
apply (rule strong_replacementI)
apply (rule_tac u="{B,A,n,0,Memrel(succ(n))}"
         in gen_separation_multi [OF list_replacement1_Reflects],
       auto simp add: nonempty)
apply (rule_tac env="[B,A,n,0,Memrel(succ(n))]" in DPow_LsetI)
apply (rule sep_rules is_nat_case_iff_sats list_functor_iff_sats
            is_wfrec_iff_sats iterates_MH_iff_sats quasinat_iff_sats |
simp)+
done


lemma list_replacement2_Reflects:
 "REFLECTS
   [λx. ∃u[L]. u ∈ B & u ∈ nat &
               is_iterates(L, is_list_functor(L, A), 0, u, x),
    λi x. ∃u ∈ Lset(i). u ∈ B & u ∈ nat &
```

```
             is_iterates(##Lset(i), is_list_functor(##Lset(i), A), 0,
u, x)]"
by (intro FOL_reflections
          is_iterates_reflection list_functor_reflection)
```

```
lemma list_replacement2:
   "L(A) ==> strong_replacement(L,
         λn y. n∈nat & is_iterates(L, is_list_functor(L,A), 0, n, y))"
apply (rule strong_replacementI)
apply (rule_tac u="{A,B,0,nat}"
         in gen_separation_multi [OF list_replacement2_Reflects],
       auto simp add: L_nat nonempty)
apply (rule_tac env="[A,B,0,nat]" in DPow_LsetI)
apply (rule sep_rules list_functor_iff_sats is_iterates_iff_sats | simp)+
done
```

12.3 L is Closed Under the Operator formula

12.3.1 Instances of Replacement for Formulas

```
lemma formula_replacement1_Reflects:
 "REFLECTS
   [λx. ∃u[L]. u ∈ B & (∃y[L]. pair(L,u,y,x) &
         is_wfrec(L, iterates_MH(L, is_formula_functor(L), 0), memsn,
u, y)),
    λi x. ∃u ∈ Lset(i). u ∈ B & (∃y ∈ Lset(i). pair(##Lset(i), u, y,
x) &
         is_wfrec(##Lset(i),
                  iterates_MH(##Lset(i),
                          is_formula_functor(##Lset(i)), 0), memsn, u,
y))]"
by (intro FOL_reflections function_reflections is_wfrec_reflection
          iterates_MH_reflection formula_functor_reflection)
```

```
lemma formula_replacement1:
   "iterates_replacement(L, is_formula_functor(L), 0)"
apply (unfold iterates_replacement_def wfrec_replacement_def, clarify)
apply (rule strong_replacementI)
apply (rule_tac u="{B,n,0,Memrel(succ(n))}"
         in gen_separation_multi [OF formula_replacement1_Reflects],
       auto simp add: nonempty)
apply (rule_tac env="[n,B,0,Memrel(succ(n))]" in DPow_LsetI)
apply (rule sep_rules is_nat_case_iff_sats formula_functor_iff_sats
            is_wfrec_iff_sats iterates_MH_iff_sats quasinat_iff_sats |
simp)+
done
```

```
lemma formula_replacement2_Reflects:
 "REFLECTS
   [λx. ∃u[L]. u ∈ B & u ∈ nat &
```

```
               is_iterates(L, is_formula_functor(L), 0, u, x),
    λi x. ∃u ∈ Lset(i). u ∈ B & u ∈ nat &
               is_iterates(##Lset(i), is_formula_functor(##Lset(i)), 0,
u, x)]"
by (intro FOL_reflections
          is_iterates_reflection formula_functor_reflection)

lemma formula_replacement2:
   "strong_replacement(L,
         λn y. n∈nat & is_iterates(L, is_formula_functor(L), 0, n, y))"
apply (rule strong_replacementI)
apply (rule_tac u="{B,0,nat}"
         in gen_separation_multi [OF formula_replacement2_Reflects],
       auto simp add: nonempty L_nat)
apply (rule_tac env="[B,0,nat]" in DPow_LsetI)
apply (rule sep_rules formula_functor_iff_sats is_iterates_iff_sats |
simp)+
done
```

NB The proofs for type `formula` are virtually identical to those for `list(A)`. It was a cut-and-paste job!

12.3.2 The Formula `is_nth`, Internalized

```
definition
  nth_fm :: "[i,i,i]=>i" where
    "nth_fm(n,l,Z) ==
       Exists(And(is_iterates_fm(tl_fm(1,0), succ(l), succ(n), 0),
              hd_fm(0,succ(Z))))"

lemma nth_fm_type [TC]:
 "[| x ∈ nat; y ∈ nat; z ∈ nat |] ==> nth_fm(x,y,z) ∈ formula"
by (simp add: nth_fm_def)

lemma sats_nth_fm [simp]:
   "[| x < length(env); y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, nth_fm(x,y,z), env) <->
        is_nth(##A, nth(x,env), nth(y,env), nth(z,env))"
apply (frule lt_length_in_nat, assumption)
apply (simp add: nth_fm_def is_nth_def sats_is_iterates_fm)
done

lemma nth_iff_sats:
      "[| nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
          i < length(env); j ∈ nat; k ∈ nat; env ∈ list(A)|]
       ==> is_nth(##A, x, y, z) <-> sats(A, nth_fm(i,j,k), env)"
by (simp add: sats_nth_fm)

theorem nth_reflection:
```

```
    "REFLECTS[λx. is_nth(L, f(x), g(x), h(x)),
              λi x. is_nth(##Lset(i), f(x), g(x), h(x))]"
apply (simp only: is_nth_def)
apply (intro FOL_reflections is_iterates_reflection
            hd_reflection tl_reflection)
done
```

12.3.3 An Instance of Replacement for *nth*

```
lemma nth_replacement_Reflects:
 "REFLECTS
   [λx. ∃u[L]. u ∈ B & (∃y[L]. pair(L,u,y,x) &
         is_wfrec(L, iterates_MH(L, is_tl(L), z), memsn, u, y)),
    λi x. ∃u ∈ Lset(i). u ∈ B & (∃y ∈ Lset(i). pair(##Lset(i), u, y,
x) &
         is_wfrec(##Lset(i),
                  iterates_MH(##Lset(i),
                         is_tl(##Lset(i)), z), memsn, u, y))]"
by (intro FOL_reflections function_reflections is_wfrec_reflection
          iterates_MH_reflection tl_reflection)

lemma nth_replacement:
   "L(w) ==> iterates_replacement(L, is_tl(L), w)"
apply (unfold iterates_replacement_def wfrec_replacement_def, clarify)
apply (rule strong_replacementI)
apply (rule_tac u="{B,w,Memrel(succ(n))}"
         in gen_separation_multi [OF nth_replacement_Reflects],
       auto)
apply (rule_tac env="[B,w,Memrel(succ(n))]" in DPow_LsetI)
apply (rule sep_rules is_nat_case_iff_sats tl_iff_sats
           is_wfrec_iff_sats iterates_MH_iff_sats quasinat_iff_sats |
simp)+
done
```

12.3.4 Instantiating the locale *M_datatypes*

```
lemma M_datatypes_axioms_L: "M_datatypes_axioms(L)"
  apply (rule M_datatypes_axioms.intro)
      apply (assumption | rule
        list_replacement1 list_replacement2
        formula_replacement1 formula_replacement2
        nth_replacement)+
  done

theorem M_datatypes_L: "PROP M_datatypes(L)"
  apply (rule M_datatypes.intro)
   apply (rule M_trancl_L)
  apply (rule M_datatypes_axioms_L)
  done
```

```
interpretation M_datatypes [L] by (rule M_datatypes_L)
```

12.4 *L* is Closed Under the Operator *eclose*

12.4.1 Instances of Replacement for *eclose*

```
lemma eclose_replacement1_Reflects:
 "REFLECTS
   [λx. ∃u[L]. u ∈ B & (∃y[L]. pair(L,u,y,x) &
         is_wfrec(L, iterates_MH(L, big_union(L), A), memsn, u, y)),
    λi x. ∃u ∈ Lset(i). u ∈ B & (∃y ∈ Lset(i). pair(##Lset(i), u, y,
x) &
         is_wfrec(##Lset(i),
                  iterates_MH(##Lset(i), big_union(##Lset(i)), A),
                  memsn, u, y))]"
by (intro FOL_reflections function_reflections is_wfrec_reflection
          iterates_MH_reflection)

lemma eclose_replacement1:
   "L(A) ==> iterates_replacement(L, big_union(L), A)"
apply (unfold iterates_replacement_def wfrec_replacement_def, clarify)
apply (rule strong_replacementI)
apply (rule_tac u="{B,A,n,Memrel(succ(n))}"
         in gen_separation_multi [OF eclose_replacement1_Reflects], auto)
apply (rule_tac env="[B,A,n,Memrel(succ(n))]" in DPow_LsetI)
apply (rule sep_rules iterates_MH_iff_sats is_nat_case_iff_sats
             is_wfrec_iff_sats big_union_iff_sats quasinat_iff_sats |
simp)+
done


lemma eclose_replacement2_Reflects:
 "REFLECTS
   [λx. ∃u[L]. u ∈ B & u ∈ nat &
                is_iterates(L, big_union(L), A, u, x),
    λi x. ∃u ∈ Lset(i). u ∈ B & u ∈ nat &
               is_iterates(##Lset(i), big_union(##Lset(i)), A, u, x)]"
by (intro FOL_reflections function_reflections is_iterates_reflection)

lemma eclose_replacement2:
   "L(A) ==> strong_replacement(L,
         λn y. n∈nat & is_iterates(L, big_union(L), A, n, y))"
apply (rule strong_replacementI)
apply (rule_tac u="{A,B,nat}"
         in gen_separation_multi [OF eclose_replacement2_Reflects],
       auto simp add: L_nat)
apply (rule_tac env="[A,B,nat]" in DPow_LsetI)
apply (rule sep_rules is_iterates_iff_sats big_union_iff_sats | simp)+
done
```

12.4.2 Instantiating the locale *M_eclose*

```
lemma M_eclose_axioms_L: "M_eclose_axioms(L)"
  apply (rule M_eclose_axioms.intro)
   apply (assumption | rule eclose_replacement1 eclose_replacement2)+
  done

theorem M_eclose_L: "PROP M_eclose(L)"
  apply (rule M_eclose.intro)
   apply (rule M_datatypes_L)
  apply (rule M_eclose_axioms_L)
  done

interpretation M_eclose [L] by (rule M_eclose_L)


end
```

13 Absoluteness for the Satisfies Relation on Formulas

theory *Satisfies_absolute* **imports** *Datatype_absolute Rec_Separation* **begin**

13.1 More Internalization

13.1.1 The Formula *is_depth*, Internalized

```
definition
  depth_fm :: "[i,i]=>i" where
  "depth_fm(p,n) ==
     Exists(Exists(Exists(
       And(formula_N_fm(n#+3,1),
         And(Neg(Member(p#+3,1)),
          And(succ_fm(n#+3,2),
           And(formula_N_fm(2,0), Member(p#+3,0))))))))"

lemma depth_fm_type [TC]:
 "[| x ∈ nat; y ∈ nat |] ==> depth_fm(x,y) ∈ formula"
by (simp add: depth_fm_def)

lemma sats_depth_fm [simp]:
   "[| x ∈ nat; y < length(env); env ∈ list(A)|]
    ==> sats(A, depth_fm(x,y), env) <->
        is_depth(##A, nth(x,env), nth(y,env))"
apply (frule_tac x=y in lt_length_in_nat, assumption)
apply (simp add: depth_fm_def is_depth_def)
done
```

```
lemma depth_iff_sats:
      "[| nth(i,env) = x; nth(j,env) = y;
          i ∈ nat; j < length(env); env ∈ list(A)|]
       ==> is_depth(##A, x, y) <-> sats(A, depth_fm(i,j), env)"
by (simp add: sats_depth_fm)

theorem depth_reflection:
     "REFLECTS[λx. is_depth(L, f(x), g(x)),
               λi x. is_depth(##Lset(i), f(x), g(x))]"
apply (simp only: is_depth_def)
apply (intro FOL_reflections function_reflections formula_N_reflection)

done
```

13.1.2 The Operator *is_formula_case*

The arguments of *is_a* are always 2, 1, 0, and the formula will be enclosed by three quantifiers.

```
definition
  formula_case_fm :: "[i, i, i, i, i, i]=>i" where
  "formula_case_fm(is_a, is_b, is_c, is_d, v, z) ==
        And(Forall(Forall(Implies(finite_ordinal_fm(1),
                           Implies(finite_ordinal_fm(0),
                            Implies(Member_fm(1,0,v#+2),
                             Forall(Implies(Equal(0,z#+3), is_a))))))),
        And(Forall(Forall(Implies(finite_ordinal_fm(1),
                           Implies(finite_ordinal_fm(0),
                            Implies(Equal_fm(1,0,v#+2),
                             Forall(Implies(Equal(0,z#+3), is_b))))))),
        And(Forall(Forall(Implies(mem_formula_fm(1),
                           Implies(mem_formula_fm(0),
                            Implies(Nand_fm(1,0,v#+2),
                             Forall(Implies(Equal(0,z#+3), is_c))))))),
        Forall(Implies(mem_formula_fm(0),
                       Implies(Forall_fm(0,succ(v)),
                             Forall(Implies(Equal(0,z#+2), is_d))))))))"


lemma is_formula_case_type [TC]:
     "[| is_a ∈ formula;  is_b ∈ formula;  is_c ∈ formula;  is_d ∈ formula;

         x ∈ nat; y ∈ nat |]
      ==> formula_case_fm(is_a, is_b, is_c, is_d, x, y) ∈ formula"
by (simp add: formula_case_fm_def)

lemma sats_formula_case_fm:
  assumes is_a_iff_sats:
      "!!a0 a1 a2.
```

```
        [|a0∈A; a1∈A; a2∈A|]
        ==> ISA(a2, a1, a0) <-> sats(A, is_a, Cons(a0,Cons(a1,Cons(a2,env))))"
  and is_b_iff_sats:
      "!!a0 a1 a2.
        [|a0∈A; a1∈A; a2∈A|]
        ==> ISB(a2, a1, a0) <-> sats(A, is_b, Cons(a0,Cons(a1,Cons(a2,env))))"
  and is_c_iff_sats:
      "!!a0 a1 a2.
        [|a0∈A; a1∈A; a2∈A|]
        ==> ISC(a2, a1, a0) <-> sats(A, is_c, Cons(a0,Cons(a1,Cons(a2,env))))"
  and is_d_iff_sats:
      "!!a0 a1.
        [|a0∈A; a1∈A|]
        ==> ISD(a1, a0) <-> sats(A, is_d, Cons(a0,Cons(a1,env)))"
  shows
      "[|x ∈ nat; y < length(env); env ∈ list(A)|]
       ==> sats(A, formula_case_fm(is_a,is_b,is_c,is_d,x,y), env) <->
           is_formula_case(##A, ISA, ISB, ISC, ISD, nth(x,env), nth(y,env))"
apply (frule_tac x=y in lt_length_in_nat, assumption)
apply (simp add: formula_case_fm_def is_formula_case_def
                 is_a_iff_sats [THEN iff_sym] is_b_iff_sats [THEN iff_sym]
                 is_c_iff_sats [THEN iff_sym] is_d_iff_sats [THEN iff_sym])
done

lemma formula_case_iff_sats:
  assumes is_a_iff_sats:
      "!!a0 a1 a2.
        [|a0∈A; a1∈A; a2∈A|]
        ==> ISA(a2, a1, a0) <-> sats(A, is_a, Cons(a0,Cons(a1,Cons(a2,env))))"
  and is_b_iff_sats:
      "!!a0 a1 a2.
        [|a0∈A; a1∈A; a2∈A|]
        ==> ISB(a2, a1, a0) <-> sats(A, is_b, Cons(a0,Cons(a1,Cons(a2,env))))"
  and is_c_iff_sats:
      "!!a0 a1 a2.
        [|a0∈A; a1∈A; a2∈A|]
        ==> ISC(a2, a1, a0) <-> sats(A, is_c, Cons(a0,Cons(a1,Cons(a2,env))))"
  and is_d_iff_sats:
      "!!a0 a1.
        [|a0∈A; a1∈A|]
        ==> ISD(a1, a0) <-> sats(A, is_d, Cons(a0,Cons(a1,env)))"
  shows
      "[|nth(i,env) = x; nth(j,env) = y;
      i ∈ nat; j < length(env); env ∈ list(A)|]
       ==> is_formula_case(##A, ISA, ISB, ISC, ISD, x, y) <->
           sats(A, formula_case_fm(is_a,is_b,is_c,is_d,i,j), env)"
by (simp add: sats_formula_case_fm [OF is_a_iff_sats is_b_iff_sats
                                       is_c_iff_sats is_d_iff_sats])
```

The second argument of `is_a` gives it direct access to `x`, which is essential for handling free variable references. Treatment is based on that of *is_nat_case_reflection*.

```
theorem is_formula_case_reflection:
  assumes is_a_reflection:
    "!!h f g g'. REFLECTS[λx. is_a(L, h(x), f(x), g(x), g'(x)),
                     λi x. is_a(##Lset(i), h(x), f(x), g(x), g'(x))]"
  and is_b_reflection:
    "!!h f g g'. REFLECTS[λx. is_b(L, h(x), f(x), g(x), g'(x)),
                     λi x. is_b(##Lset(i), h(x), f(x), g(x), g'(x))]"
  and is_c_reflection:
    "!!h f g g'. REFLECTS[λx. is_c(L, h(x), f(x), g(x), g'(x)),
                     λi x. is_c(##Lset(i), h(x), f(x), g(x), g'(x))]"
  and is_d_reflection:
    "!!h f g g'. REFLECTS[λx. is_d(L, h(x), f(x), g(x)),
                     λi x. is_d(##Lset(i), h(x), f(x), g(x))]"
  shows "REFLECTS[λx. is_formula_case(L, is_a(L,x), is_b(L,x), is_c(L,x),
is_d(L,x), g(x), h(x)),
               λi x. is_formula_case(##Lset(i), is_a(##Lset(i), x), is_b(##Lset(i),
x), is_c(##Lset(i), x), is_d(##Lset(i), x), g(x), h(x))]"
apply (simp (no_asm_use) only: is_formula_case_def)
apply (intro FOL_reflections function_reflections finite_ordinal_reflection
         mem_formula_reflection
         Member_reflection Equal_reflection Nand_reflection Forall_reflection
         is_a_reflection is_b_reflection is_c_reflection is_d_reflection)
done
```

13.2 Absoluteness for the Function *satisfies*

```
definition
  is_depth_apply :: "[i=>o,i,i,i] => o" where
   — Merely a useful abbreviation for the sequel.
  "is_depth_apply(M,h,p,z) ==
    ∃dp[M]. ∃sdp[M]. ∃hsdp[M].
        finite_ordinal(M,dp) & is_depth(M,p,dp) & successor(M,dp,sdp)
&
        fun_apply(M,h,sdp,hsdp) & fun_apply(M,hsdp,p,z)"

lemma (in M_datatypes) is_depth_apply_abs [simp]:
     "[|M(h); p ∈ formula; M(z)|]
      ==> is_depth_apply(M,h,p,z) <-> z = h ` succ(depth(p)) ` p"
by (simp add: is_depth_apply_def formula_into_M depth_type eq_commute)
```

There is at present some redundancy between the relativizations in e.g. *satisfies_is_a* and those in e.g. *Member_replacement*.

These constants let us instantiate the parameters *a*, *b*, *c*, *d*, etc., of the locale *Formula_Rec*.

```
definition
  satisfies_a :: "[i,i,i]=>i" where
```

```
   "satisfies_a(A) ==
    λx y. λenv ∈ list(A). bool_of_o (nth(x,env) ∈ nth(y,env))"
```

definition

```
  satisfies_is_a :: "[i=>o,i,i,i,i]=>o"
```
where
```
   "satisfies_is_a(M,A) ==
    λx y zz. ∀lA[M]. is_list(M,A,lA) -->
             is_lambda(M, lA,
                λenv z. is_bool_of_o(M,
                      ∃nx[M]. ∃ny[M].
                       is_nth(M,x,env,nx) & is_nth(M,y,env,ny) & nx ∈
ny, z),
                zz)"
```

definition

```
  satisfies_b :: "[i,i,i]=>i"
```
where
```
   "satisfies_b(A) ==
    λx y. λenv ∈ list(A). bool_of_o (nth(x,env) = nth(y,env))"
```

definition

```
  satisfies_is_b :: "[i=>o,i,i,i,i]=>o"
```
where

— We simplify the formula to have just `nx` rather than introducing `ny` with `nx
= ny`

```
  "satisfies_is_b(M,A) ==
    λx y zz. ∀lA[M]. is_list(M,A,lA) -->
             is_lambda(M, lA,
                λenv z. is_bool_of_o(M,
                      ∃nx[M]. is_nth(M,x,env,nx) & is_nth(M,y,env,nx),
z),
                zz)"
```

definition

```
  satisfies_c :: "[i,i,i,i,i]=>i"
```
where
```
   "satisfies_c(A) == λp q rp rq. λenv ∈ list(A). not(rp ‘ env and rq
‘ env)"
```

definition

```
  satisfies_is_c :: "[i=>o,i,i,i,i,i]=>o"
```
where
```
   "satisfies_is_c(M,A,h) ==
    λp q zz. ∀lA[M]. is_list(M,A,lA) -->
             is_lambda(M, lA, λenv z. ∃hp[M]. ∃hq[M].
                 (∃rp[M]. is_depth_apply(M,h,p,rp) & fun_apply(M,rp,env,hp))
&
                 (∃rq[M]. is_depth_apply(M,h,q,rq) & fun_apply(M,rq,env,hq))
&
                 (∃pq[M]. is_and(M,hp,hq,pq) & is_not(M,pq,z)),
                zz)"
```

definition

```
  satisfies_d :: "[i,i,i]=>i" where
   "satisfies_d(A)
    == λp rp. λenv ∈ list(A). bool_of_o (∀x∈A. rp ` (Cons(x,env)) =
1)"
```

definition

```
  satisfies_is_d :: "[i=>o,i,i,i,i]=>o" where
   "satisfies_is_d(M,A,h) ==
    λp zz. ∀lA[M]. is_list(M,A,lA) -->
             is_lambda(M, lA,
                λenv z. ∃rp[M]. is_depth_apply(M,h,p,rp) &
                    is_bool_of_o(M,
                           ∀x[M]. ∀xenv[M]. ∀hp[M].
                              x∈A --> is_Cons(M,x,env,xenv) -->
                              fun_apply(M,rp,xenv,hp) --> number1(M,hp),
                  z),
               zz)"
```

definition

```
  satisfies_MH :: "[i=>o,i,i,i,i]=>o" where
    — The variable u is unused, but gives satisfies_MH the correct arity.
  "satisfies_MH ==
    λM A u f z.
         ∀fml[M]. is_formula(M,fml) -->
             is_lambda (M, fml,
               is_formula_case (M, satisfies_is_a(M,A),
                                satisfies_is_b(M,A),
                                satisfies_is_c(M,A,f), satisfies_is_d(M,A,f)),
               z)"
```

definition

```
  is_satisfies :: "[i=>o,i,i,i]=>o" where
  "is_satisfies(M,A) == is_formula_rec (M, satisfies_MH(M,A))"
```

This lemma relates the fragments defined above to the original primitive recursion in *satisfies*. Induction is not required: the definitions are directly equal!

lemma *satisfies_eq:*

```
  "satisfies(A,p) =
   formula_rec (satisfies_a(A), satisfies_b(A),
                satisfies_c(A), satisfies_d(A), p)"
```

by *(simp add: satisfies_formula_def satisfies_a_def satisfies_b_def satisfies_c_def satisfies_d_def)*

Further constraints on the class *M* in order to prove absoluteness for the constants defined above. The ultimate goal is the absoluteness of the function *satisfies*.

locale *M_satisfies = M_eclose +*
 assumes

```
  Member_replacement:
   "[|M(A); x ∈ nat; y ∈ nat|]
    ==> strong_replacement
        (M, λenv z. ∃bo[M]. ∃nx[M]. ∃ny[M].
             env ∈ list(A) & is_nth(M,x,env,nx) & is_nth(M,y,env,ny)
&
             is_bool_of_o(M, nx ∈ ny, bo) &
             pair(M, env, bo, z))"
```

and

```
  Equal_replacement:
   "[|M(A); x ∈ nat; y ∈ nat|]
    ==> strong_replacement
        (M, λenv z. ∃bo[M]. ∃nx[M]. ∃ny[M].
             env ∈ list(A) & is_nth(M,x,env,nx) & is_nth(M,y,env,ny)
&
             is_bool_of_o(M, nx = ny, bo) &
             pair(M, env, bo, z))"
```

and

```
  Nand_replacement:
   "[|M(A); M(rp); M(rq)|]
    ==> strong_replacement
        (M, λenv z. ∃rpe[M]. ∃rqe[M]. ∃andpq[M]. ∃notpq[M].
              fun_apply(M,rp,env,rpe) & fun_apply(M,rq,env,rqe) &
              is_and(M,rpe,rqe,andpq) & is_not(M,andpq,notpq) &
              env ∈ list(A) & pair(M, env, notpq, z))"
```

and

```
 Forall_replacement:
  "[|M(A); M(rp)|]
   ==> strong_replacement
       (M, λenv z. ∃bo[M].
             env ∈ list(A) &
             is_bool_of_o (M,
                          ∀a[M]. ∀co[M]. ∀rpco[M].
                             a∈A --> is_Cons(M,a,env,co) -->
                             fun_apply(M,rp,co,rpco) --> number1(M,
rpco),
                          bo) &
             pair(M,env,bo,z))"
```

and

`formula_rec_replacement:`

— For the *transrec*

```
  "[|n ∈ nat; M(A)|] ==> transrec_replacement(M, satisfies_MH(M,A), n)"
```

and

`formula_rec_lambda_replacement:`

— For the *λ-abstraction* in the *transrec* body

```
  "[|M(g); M(A)|] ==>
   strong_replacement (M,
      λx y. mem_formula(M,x) &
            (∃c[M]. is_formula_case(M, satisfies_is_a(M,A),
```

```
                                satisfies_is_b(M,A),
                                satisfies_is_c(M,A,g),
                                satisfies_is_d(M,A,g), x, c) &
            pair(M, x, c, y)))"


lemma (in M_satisfies) Member_replacement':
    "[|M(A); x ∈ nat; y ∈ nat|]
     ==> strong_replacement
         (M, λenv z. env ∈ list(A) &
                    z = ⟨env, bool_of_o(nth(x, env) ∈ nth(y, env))⟩)"
by (insert Member_replacement, simp)

lemma (in M_satisfies) Equal_replacement':
    "[|M(A); x ∈ nat; y ∈ nat|]
     ==> strong_replacement
         (M, λenv z. env ∈ list(A) &
                    z = ⟨env, bool_of_o(nth(x, env) = nth(y, env))⟩)"
by (insert Equal_replacement, simp)

lemma (in M_satisfies) Nand_replacement':
    "[|M(A); M(rp); M(rq)|]
     ==> strong_replacement
         (M, λenv z. env ∈ list(A) & z = ⟨env, not(rp`env and rq`env)⟩)"
by (insert Nand_replacement, simp)

lemma (in M_satisfies) Forall_replacement':
   "[|M(A); M(rp)|]
    ==> strong_replacement
        (M, λenv z.
               env ∈ list(A) &
               z = ⟨env, bool_of_o (∀a∈A. rp ` Cons(a,env) = 1)⟩)"
by (insert Forall_replacement, simp)

lemma (in M_satisfies) a_closed:
     "[|M(A); x∈nat; y∈nat|] ==> M(satisfies_a(A,x,y))"
apply (simp add: satisfies_a_def)
apply (blast intro: lam_closed2 Member_replacement')
done

lemma (in M_satisfies) a_rel:
     "M(A) ==> Relation2(M, nat, nat, satisfies_is_a(M,A), satisfies_a(A))"
apply (simp add: Relation2_def satisfies_is_a_def satisfies_a_def)
apply (auto del: iffI intro!: lambda_abs2 simp add: Relation1_def)
done

lemma (in M_satisfies) b_closed:
     "[|M(A); x∈nat; y∈nat|] ==> M(satisfies_b(A,x,y))"
apply (simp add: satisfies_b_def)
```

```
apply (blast intro: lam_closed2 Equal_replacement')
done

lemma (in M_satisfies) b_rel:
     "M(A) ==> Relation2(M, nat, nat, satisfies_is_b(M,A), satisfies_b(A))"
apply (simp add: Relation2_def satisfies_is_b_def satisfies_b_def)
apply (auto del: iffI intro!: lambda_abs2 simp add: Relation1_def)
done

lemma (in M_satisfies) c_closed:
     "[|M(A); x ∈ formula; y ∈ formula; M(rx); M(ry)|]
      ==> M(satisfies_c(A,x,y,rx,ry))"
apply (simp add: satisfies_c_def)
apply (rule lam_closed2)
apply (rule Nand_replacement')
apply (simp_all add: formula_into_M list_into_M [of _ A])
done

lemma (in M_satisfies) c_rel:
 "[|M(A); M(f)|] ==>
  Relation2 (M, formula, formula,
               satisfies_is_c(M,A,f),
               λu v. satisfies_c(A, u, v, f ` succ(depth(u)) ` u,
                                           f ` succ(depth(v)) ` v))"
apply (simp add: Relation2_def satisfies_is_c_def satisfies_c_def)
apply (auto del: iffI intro!: lambda_abs2
            simp add: Relation1_def formula_into_M)
done

lemma (in M_satisfies) d_closed:
     "[|M(A); x ∈ formula; M(rx)|] ==> M(satisfies_d(A,x,rx))"
apply (simp add: satisfies_d_def)
apply (rule lam_closed2)
apply (rule Forall_replacement')
apply (simp_all add: formula_into_M list_into_M [of _ A])
done

lemma (in M_satisfies) d_rel:
 "[|M(A); M(f)|] ==>
  Relation1(M, formula, satisfies_is_d(M,A,f),
     λu. satisfies_d(A, u, f ` succ(depth(u)) ` u))"
apply (simp del: rall_abs
            add: Relation1_def satisfies_is_d_def satisfies_d_def)
apply (auto del: iffI intro!: lambda_abs2 simp add: Relation1_def)
done


lemma (in M_satisfies) fr_replace:
      "[|n ∈ nat; M(A)|] ==> transrec_replacement(M,satisfies_MH(M,A),n)"
```

```
by (blast intro: formula_rec_replacement)

lemma (in M_satisfies) formula_case_satisfies_closed:
 "[|M(g); M(A); x ∈ formula|] ==>
  M(formula_case (satisfies_a(A), satisfies_b(A),
       λu v. satisfies_c(A, u, v,
                         g ‘ succ(depth(u)) ‘ u, g ‘ succ(depth(v)) ‘
v),
       λu. satisfies_d (A, u, g ‘ succ(depth(u)) ‘ u),
       x))"
by (blast intro: formula_case_closed a_closed b_closed c_closed d_closed)


lemma (in M_satisfies) fr_lam_replace:
   "[|M(g); M(A)|] ==>
    strong_replacement (M, λx y. x ∈ formula &
            y = ⟨x,
                 formula_rec_case(satisfies_a(A),
                                  satisfies_b(A),
                                  satisfies_c(A),
                                  satisfies_d(A), g, x)⟩)"
apply (insert formula_rec_lambda_replacement)
apply (simp add: formula_rec_case_def formula_case_satisfies_closed
                 formula_case_abs [OF a_rel b_rel c_rel d_rel])
done
```

Instantiate locale `Formula_Rec` for the Function `satisfies`

```
lemma (in M_satisfies) Formula_Rec_axioms_M:
   "M(A) ==>
    Formula_Rec_axioms(M, satisfies_a(A), satisfies_is_a(M,A),
                          satisfies_b(A), satisfies_is_b(M,A),
                          satisfies_c(A), satisfies_is_c(M,A),
                          satisfies_d(A), satisfies_is_d(M,A))"
apply (rule Formula_Rec_axioms.intro)
apply (assumption |
       rule a_closed a_rel b_closed b_rel c_closed c_rel d_closed d_rel
       fr_replace [unfolded satisfies_MH_def]
       fr_lam_replace) +
done


theorem (in M_satisfies) Formula_Rec_M:
    "M(A) ==>
     PROP Formula_Rec(M, satisfies_a(A), satisfies_is_a(M,A),
                         satisfies_b(A), satisfies_is_b(M,A),
                         satisfies_c(A), satisfies_is_c(M,A),
                         satisfies_d(A), satisfies_is_d(M,A))"
  apply (rule Formula_Rec.intro)
```

```
  apply (rule M_satisfies.axioms, rule M_satisfies_axioms)
 apply (erule Formula_Rec_axioms_M)
 done

lemmas (in M_satisfies)
   satisfies_closed' = Formula_Rec.formula_rec_closed [OF Formula_Rec_M]
and satisfies_abs'    = Formula_Rec.formula_rec_abs [OF Formula_Rec_M]


lemma (in M_satisfies) satisfies_closed:
  "[|M(A); p ∈ formula|] ==> M(satisfies(A,p))"
by (simp add: Formula_Rec.formula_rec_closed [OF Formula_Rec_M]
             satisfies_eq)

lemma (in M_satisfies) satisfies_abs:
  "[|M(A); M(z); p ∈ formula|]
   ==> is_satisfies(M,A,p,z) <-> z = satisfies(A,p)"
by (simp only: Formula_Rec.formula_rec_abs [OF Formula_Rec_M]
              satisfies_eq is_satisfies_def satisfies_MH_def)
```

13.3 Internalizations Needed to Instantiate *M_satisfies*

13.3.1 The Operator *is_depth_apply*, Internalized

```
definition
  depth_apply_fm :: "[i,i,i]=>i" where
    "depth_apply_fm(h,p,z) ==
       Exists(Exists(Exists(
        And(finite_ordinal_fm(2),
         And(depth_fm(p#+3,2),
          And(succ_fm(2,1),
           And(fun_apply_fm(h#+3,1,0), fun_apply_fm(0,p#+3,z#+3)))))))"

lemma depth_apply_type [TC]:
     "[| x ∈ nat; y ∈ nat; z ∈ nat |] ==> depth_apply_fm(x,y,z) ∈ formula"
by (simp add: depth_apply_fm_def)

lemma sats_depth_apply_fm [simp]:
   "[| x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, depth_apply_fm(x,y,z), env) <->
        is_depth_apply(##A, nth(x,env), nth(y,env), nth(z,env))"
by (simp add: depth_apply_fm_def is_depth_apply_def)

lemma depth_apply_iff_sats:
    "[| nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
        i ∈ nat; j ∈ nat; k ∈ nat; env ∈ list(A)|]
     ==> is_depth_apply(##A, x, y, z) <-> sats(A, depth_apply_fm(i,j,k),
env)"
by simp
```

lemma *depth_apply_reflection:*
```
     "REFLECTS[λx. is_depth_apply(L,f(x),g(x),h(x)),
               λi x. is_depth_apply(##Lset(i),f(x),g(x),h(x))]"
```
apply *(simp only: is_depth_apply_def)*
apply *(intro FOL_reflections function_reflections depth_reflection*
finite_ordinal_reflection)
done

13.3.2 The Operator *satisfies_is_a*, Internalized

definition
```
  satisfies_is_a_fm :: "[i,i,i,i]=>i" where
  "satisfies_is_a_fm(A,x,y,z) ==
   Forall(
     Implies(is_list_fm(succ(A),0),
       lambda_fm(
         bool_of_o_fm(Exists(
                       Exists(And(nth_fm(x#+6,3,1),
                               And(nth_fm(y#+6,3,0),
                                   Member(1,0))))), 0),
         0, succ(z))))"
```

lemma *satisfies_is_a_type [TC]:*
```
     "[| A ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat |]
      ==> satisfies_is_a_fm(A,x,y,z) ∈ formula"
```
by *(simp add: satisfies_is_a_fm_def)*

lemma *sats_satisfies_is_a_fm [simp]:*
```
   "[| u ∈ nat; x < length(env); y < length(env); z ∈ nat; env ∈ list(A)|]
    ==> sats(A, satisfies_is_a_fm(u,x,y,z), env) <->
        satisfies_is_a(##A, nth(u,env), nth(x,env), nth(y,env), nth(z,env))"
```
apply *(frule_tac x=x* **in** *lt_length_in_nat, assumption)*
apply *(frule_tac x=y* **in** *lt_length_in_nat, assumption)*
apply *(simp add: satisfies_is_a_fm_def satisfies_is_a_def sats_lambda_fm*

sats_bool_of_o_fm)
done

lemma *satisfies_is_a_iff_sats:*
```
  "[| nth(u,env) = nu; nth(x,env) = nx; nth(y,env) = ny; nth(z,env) =
nz;
      u ∈ nat; x < length(env); y < length(env); z ∈ nat; env ∈ list(A)|]
   ==> satisfies_is_a(##A,nu,nx,ny,nz) <->
       sats(A, satisfies_is_a_fm(u,x,y,z), env)"
```
by *simp*

theorem *satisfies_is_a_reflection:*
```
     "REFLECTS[λx. satisfies_is_a(L,f(x),g(x),h(x),g'(x)),
               λi x. satisfies_is_a(##Lset(i),f(x),g(x),h(x),g'(x))]"
```

```
apply (unfold satisfies_is_a_def)
apply (intro FOL_reflections is_lambda_reflection bool_of_o_reflection

             nth_reflection is_list_reflection)
done
```

13.3.3 The Operator *satisfies_is_b*, Internalized

```
definition
  satisfies_is_b_fm :: "[i,i,i,i]=>i" where
 "satisfies_is_b_fm(A,x,y,z) ==
   Forall(
     Implies(is_list_fm(succ(A),0),
       lambda_fm(
         bool_of_o_fm(Exists(And(nth_fm(x#+5,2,0), nth_fm(y#+5,2,0))),
0),
         0, succ(z))))"

lemma satisfies_is_b_type [TC]:
     "[| A ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat |]
      ==> satisfies_is_b_fm(A,x,y,z) ∈ formula"
by (simp add: satisfies_is_b_fm_def)

lemma sats_satisfies_is_b_fm [simp]:
   "[| u ∈ nat; x < length(env); y < length(env); z ∈ nat; env ∈ list(A)|]
    ==> sats(A, satisfies_is_b_fm(u,x,y,z), env) <->
        satisfies_is_b(##A, nth(u,env), nth(x,env), nth(y,env), nth(z,env))"
apply (frule_tac x=x in lt_length_in_nat, assumption)
apply (frule_tac x=y in lt_length_in_nat, assumption)
apply (simp add: satisfies_is_b_fm_def satisfies_is_b_def sats_lambda_fm

                 sats_bool_of_o_fm)
done

lemma satisfies_is_b_iff_sats:
  "[| nth(u,env) = nu; nth(x,env) = nx; nth(y,env) = ny; nth(z,env) =
nz;
      u ∈ nat; x < length(env); y < length(env); z ∈ nat; env ∈ list(A)|]
   ==> satisfies_is_b(##A,nu,nx,ny,nz) <->
       sats(A, satisfies_is_b_fm(u,x,y,z), env)"
by simp

theorem satisfies_is_b_reflection:
     "REFLECTS[λx. satisfies_is_b(L,f(x),g(x),h(x),g'(x)),
               λi x. satisfies_is_b(##Lset(i),f(x),g(x),h(x),g'(x))]"
apply (unfold satisfies_is_b_def)
apply (intro FOL_reflections is_lambda_reflection bool_of_o_reflection

             nth_reflection is_list_reflection)
```

done

13.3.4 The Operator *satisfies_is_c*, Internalized

```
definition
  satisfies_is_c_fm :: "[i,i,i,i,i]=>i" where
 "satisfies_is_c_fm(A,h,p,q,zz) ==
   Forall(
     Implies(is_list_fm(succ(A),0),
       lambda_fm(
         Exists(Exists(
          And(Exists(And(depth_apply_fm(h#+7,p#+7,0), fun_apply_fm(0,4,2))),
          And(Exists(And(depth_apply_fm(h#+7,q#+7,0), fun_apply_fm(0,4,1))),
              Exists(And(and_fm(2,1,0), not_fm(0,3))))))),
         0, succ(zz))))"

lemma satisfies_is_c_type [TC]:
     "[| A ∈ nat; h ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat |]
      ==> satisfies_is_c_fm(A,h,x,y,z) ∈ formula"
by (simp add: satisfies_is_c_fm_def)

lemma sats_satisfies_is_c_fm [simp]:
   "[| u ∈ nat; v ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, satisfies_is_c_fm(u,v,x,y,z), env) <->
        satisfies_is_c(##A, nth(u,env), nth(v,env), nth(x,env),
                            nth(y,env), nth(z,env))"
by (simp add: satisfies_is_c_fm_def satisfies_is_c_def sats_lambda_fm)

lemma satisfies_is_c_iff_sats:
  "[| nth(u,env) = nu; nth(v,env) = nv; nth(x,env) = nx; nth(y,env) =
ny;
      nth(z,env) = nz;
      u ∈ nat; v ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
   ==> satisfies_is_c(##A,nu,nv,nx,ny,nz) <->
       sats(A, satisfies_is_c_fm(u,v,x,y,z), env)"
by simp

theorem satisfies_is_c_reflection:
     "REFLECTS[λx. satisfies_is_c(L,f(x),g(x),h(x),g'(x),h'(x)),
               λi x. satisfies_is_c(##Lset(i),f(x),g(x),h(x),g'(x),h'(x))]"
apply (unfold satisfies_is_c_def)
apply (intro FOL_reflections function_reflections is_lambda_reflection
             extra_reflections nth_reflection depth_apply_reflection
             is_list_reflection)
done
```

13.3.5 The Operator *satisfies_is_d*, Internalized

```
definition
  satisfies_is_d_fm :: "[i,i,i,i]=>i" where
```

```
 "satisfies_is_d_fm(A,h,p,zz) ==
   Forall(
     Implies(is_list_fm(succ(A),0),
       lambda_fm(
         Exists(
           And(depth_apply_fm(h#+5,p#+5,0),
               bool_of_o_fm(
                Forall(Forall(Forall(
                  Implies(Member(2,A#+8),
                   Implies(Cons_fm(2,5,1),
                    Implies(fun_apply_fm(3,1,0), number1_fm(0))))))), 1))),
         0, succ(zz))))"
```

lemma *satisfies_is_d_type [TC]:*
```
     "[| A ∈ nat; h ∈ nat; x ∈ nat; z ∈ nat |]
      ==> satisfies_is_d_fm(A,h,x,z) ∈ formula"
```
by *(simp add: satisfies_is_d_fm_def)*

lemma *sats_satisfies_is_d_fm [simp]:*
```
   "[| u ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, satisfies_is_d_fm(u,x,y,z), env) <->
        satisfies_is_d(##A, nth(u,env), nth(x,env), nth(y,env), nth(z,env))"
```

by *(simp add: satisfies_is_d_fm_def satisfies_is_d_def sats_lambda_fm sats_bool_of_o_fm)*

lemma *satisfies_is_d_iff_sats:*
```
  "[| nth(u,env) = nu; nth(x,env) = nx; nth(y,env) = ny; nth(z,env) =
nz;
      u ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
   ==> satisfies_is_d(##A,nu,nx,ny,nz) <->
       sats(A, satisfies_is_d_fm(u,x,y,z), env)"
```
by *simp*

theorem *satisfies_is_d_reflection:*
```
     "REFLECTS[λx. satisfies_is_d(L,f(x),g(x),h(x),g'(x)),
               λi x. satisfies_is_d(##Lset(i),f(x),g(x),h(x),g'(x))]"
```
apply *(unfold satisfies_is_d_def)*
apply *(intro FOL_reflections function_reflections is_lambda_reflection extra_reflections nth_reflection depth_apply_reflection is_list_reflection)*
done

13.3.6 The Operator *satisfies_MH*, Internalized

definition
```
  satisfies_MH_fm :: "[i,i,i,i]=>i" where
 "satisfies_MH_fm(A,u,f,zz) ==
   Forall(
```

```
      Implies(is_formula_fm(0),
        lambda_fm(
          formula_case_fm(satisfies_is_a_fm(A#+7,2,1,0),
                          satisfies_is_b_fm(A#+7,2,1,0),
                          satisfies_is_c_fm(A#+7,f#+7,2,1,0),
                          satisfies_is_d_fm(A#+6,f#+6,1,0),
                          1, 0),
          0, succ(zz))))"
```

lemma *satisfies_MH_type [TC]:*
```
      "[| A ∈ nat; u ∈ nat; x ∈ nat; z ∈ nat |]
       ==> satisfies_MH_fm(A,u,x,z) ∈ formula"
```
by *(simp add: satisfies_MH_fm_def)*

lemma *sats_satisfies_MH_fm [simp]:*
```
   "[| u ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, satisfies_MH_fm(u,x,y,z), env) <->
        satisfies_MH(##A, nth(u,env), nth(x,env), nth(y,env), nth(z,env))"
```

by *(simp add: satisfies_MH_fm_def satisfies_MH_def sats_lambda_fm*
sats_formula_case_fm)

lemma *satisfies_MH_iff_sats:*
```
  "[| nth(u,env) = nu; nth(x,env) = nx; nth(y,env) = ny; nth(z,env) =
nz;
      u ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
   ==> satisfies_MH(##A,nu,nx,ny,nz) <->
       sats(A, satisfies_MH_fm(u,x,y,z), env)"
```
by *simp*

lemmas *satisfies_reflections =*
```
       is_lambda_reflection is_formula_reflection
       is_formula_case_reflection
       satisfies_is_a_reflection satisfies_is_b_reflection
       satisfies_is_c_reflection satisfies_is_d_reflection
```

theorem *satisfies_MH_reflection:*
```
     "REFLECTS[λx. satisfies_MH(L,f(x),g(x),h(x),g'(x)),
               λi x. satisfies_MH(##Lset(i),f(x),g(x),h(x),g'(x))]"
```
apply *(unfold satisfies_MH_def)*
apply *(intro FOL_reflections satisfies_reflections)*
done

13.4 Lemmas for Instantiating the Locale *M_satisfies*

13.4.1 The *Member* Case

lemma *Member_Reflects:*
```
 "REFLECTS[λu. ∃v[L]. v ∈ B ∧ (∃bo[L]. ∃nx[L]. ∃ny[L].
          v ∈ lstA ∧ is_nth(L,x,v,nx) ∧ is_nth(L,y,v,ny) ∧
```

```
          is_bool_of_o(L, nx ∈ ny, bo) ∧ pair(L,v,bo,u)),
   λi u. ∃v ∈ Lset(i). v ∈ B ∧ (∃bo ∈ Lset(i). ∃nx ∈ Lset(i). ∃ny
∈ Lset(i).
             v ∈ lstA ∧ is_nth(##Lset(i), x, v, nx) ∧
             is_nth(##Lset(i), y, v, ny) ∧
          is_bool_of_o(##Lset(i), nx ∈ ny, bo) ∧ pair(##Lset(i), v, bo,
u))]"
by (intro FOL_reflections function_reflections nth_reflection
          bool_of_o_reflection)


lemma Member_replacement:
    "[|L(A); x ∈ nat; y ∈ nat|]
     ==> strong_replacement
         (L, λenv z. ∃bo[L]. ∃nx[L]. ∃ny[L].
              env ∈ list(A) & is_nth(L,x,env,nx) & is_nth(L,y,env,ny)
&
              is_bool_of_o(L, nx ∈ ny, bo) &
              pair(L, env, bo, z))"
apply (rule strong_replacementI)
apply (rule_tac u="{list(A),B,x,y}"
         in gen_separation_multi [OF Member_Reflects],
       auto simp add: nat_into_M list_closed)
apply (rule_tac env="[list(A),B,x,y]" in DPow_LsetI)
apply (rule sep_rules nth_iff_sats is_bool_of_o_iff_sats | simp)+
done
```

13.4.2 The *Equal* Case

```
lemma Equal_Reflects:
 "REFLECTS[λu. ∃v[L]. v ∈ B ∧ (∃bo[L]. ∃nx[L]. ∃ny[L].
          v ∈ lstA ∧ is_nth(L, x, v, nx) ∧ is_nth(L, y, v, ny) ∧
          is_bool_of_o(L, nx = ny, bo) ∧ pair(L, v, bo, u)),
   λi u. ∃v ∈ Lset(i). v ∈ B ∧ (∃bo ∈ Lset(i). ∃nx ∈ Lset(i). ∃ny
∈ Lset(i).
             v ∈ lstA ∧ is_nth(##Lset(i), x, v, nx) ∧
             is_nth(##Lset(i), y, v, ny) ∧
          is_bool_of_o(##Lset(i), nx = ny, bo) ∧ pair(##Lset(i), v, bo,
u))]"
by (intro FOL_reflections function_reflections nth_reflection
          bool_of_o_reflection)


lemma Equal_replacement:
    "[|L(A); x ∈ nat; y ∈ nat|]
     ==> strong_replacement
         (L, λenv z. ∃bo[L]. ∃nx[L]. ∃ny[L].
              env ∈ list(A) & is_nth(L,x,env,nx) & is_nth(L,y,env,ny)
&
```

```
             is_bool_of_o(L, nx = ny, bo) &
             pair(L, env, bo, z))"
apply (rule strong_replacementI)
apply (rule_tac u="{list(A),B,x,y}"
         in gen_separation_multi [OF Equal_Reflects],
       auto simp add: nat_into_M list_closed)
apply (rule_tac env="[list(A),B,x,y]" in DPow_LsetI)
apply (rule sep_rules nth_iff_sats is_bool_of_o_iff_sats | simp)+
done
```

13.4.3 The *Nand* Case

```
lemma Nand_Reflects:
    "REFLECTS [λx. ∃u[L]. u ∈ B ∧
               (∃rpe[L]. ∃rqe[L]. ∃andpq[L]. ∃notpq[L].
                 fun_apply(L, rp, u, rpe) ∧ fun_apply(L, rq, u, rqe) ∧
                 is_and(L, rpe, rqe, andpq) ∧ is_not(L, andpq, notpq)
∧
                 u ∈ list(A) ∧ pair(L, u, notpq, x)),
    λi x. ∃u ∈ Lset(i). u ∈ B ∧
     (∃rpe ∈ Lset(i). ∃rqe ∈ Lset(i). ∃andpq ∈ Lset(i). ∃notpq ∈ Lset(i).
       fun_apply(##Lset(i), rp, u, rpe) ∧ fun_apply(##Lset(i), rq, u,
rqe) ∧
       is_and(##Lset(i), rpe, rqe, andpq) ∧ is_not(##Lset(i), andpq, notpq)
∧
       u ∈ list(A) ∧ pair(##Lset(i), u, notpq, x))]"
apply (unfold is_and_def is_not_def)
apply (intro FOL_reflections function_reflections)
done

lemma Nand_replacement:
    "[|L(A); L(rp); L(rq)|]
     ==> strong_replacement
         (L, λenv z. ∃rpe[L]. ∃rqe[L]. ∃andpq[L]. ∃notpq[L].
               fun_apply(L,rp,env,rpe) & fun_apply(L,rq,env,rqe) &
               is_and(L,rpe,rqe,andpq) & is_not(L,andpq,notpq) &
               env ∈ list(A) & pair(L, env, notpq, z))"
apply (rule strong_replacementI)
apply (rule_tac u="{list(A),B,rp,rq}"
         in gen_separation_multi [OF Nand_Reflects],
       auto simp add: list_closed)
apply (rule_tac env="[list(A),B,rp,rq]" in DPow_LsetI)
apply (rule sep_rules is_and_iff_sats is_not_iff_sats | simp)+
done
```

13.4.4 The *Forall* Case

```
lemma Forall_Reflects:
 "REFLECTS [λx. ∃u[L]. u ∈ B ∧ (∃bo[L]. u ∈ list(A) ∧
                 is_bool_of_o (L,
```

```
    ∀a[L]. ∀co[L]. ∀rpco[L]. a ∈ A ⟶
              is_Cons(L,a,u,co) ⟶ fun_apply(L,rp,co,rpco) ⟶
              number1(L,rpco),
                         bo) ∧ pair(L,u,bo,x)),
 λi x. ∃u ∈ Lset(i). u ∈ B ∧ (∃bo ∈ Lset(i). u ∈ list(A) ∧
        is_bool_of_o (##Lset(i),
 ∀a ∈ Lset(i). ∀co ∈ Lset(i). ∀rpco ∈ Lset(i). a ∈ A ⟶
           is_Cons(##Lset(i),a,u,co) ⟶ fun_apply(##Lset(i),rp,co,rpco)
⟶
           number1(##Lset(i),rpco),
                     bo) ∧ pair(##Lset(i),u,bo,x))]"
```
apply *(unfold is_bool_of_o_def)*
apply *(intro FOL_reflections function_reflections Cons_reflection)*
done

```
lemma Forall_replacement:
   "[|L(A); L(rp)|]
    ==> strong_replacement
        (L, λenv z. ∃bo[L].
              env ∈ list(A) &
              is_bool_of_o (L,
                         ∀a[L]. ∀co[L]. ∀rpco[L].
                            a∈A --> is_Cons(L,a,env,co) -->
                            fun_apply(L,rp,co,rpco) --> number1(L,
rpco),
                         bo) &
              pair(L,env,bo,z))"
apply (rule strong_replacementI)
apply (rule_tac u="{A,list(A),B,rp}"
         in gen_separation_multi [OF Forall_Reflects],
       auto simp add: list_closed)
apply (rule_tac env="[A,list(A),B,rp]" in DPow_LsetI)
apply (rule sep_rules is_bool_of_o_iff_sats Cons_iff_sats | simp)+
done
```

13.4.5 The *transrec_replacement* Case

```
lemma formula_rec_replacement_Reflects:
 "REFLECTS [λx. ∃u[L]. u ∈ B ∧ (∃y[L]. pair(L, u, y, x) ∧
             is_wfrec (L, satisfies_MH(L,A), mesa, u, y)),
    λi x. ∃u ∈ Lset(i). u ∈ B ∧ (∃y ∈ Lset(i). pair(##Lset(i), u, y,
x) ∧
             is_wfrec (##Lset(i), satisfies_MH(##Lset(i),A), mesa, u,
y))]"
by (intro FOL_reflections function_reflections satisfies_MH_reflection

          is_wfrec_reflection)

lemma formula_rec_replacement:
```

— For the *transrec*

```
"[|n ∈ nat; L(A)|] ==> transrec_replacement(L, satisfies_MH(L,A), n)"
apply (rule transrec_replacementI, simp add: nat_into_M)
apply (rule strong_replacementI)
apply (rule_tac u="{B,A,n,Memrel(eclose({n}))}"
        in gen_separation_multi [OF formula_rec_replacement_Reflects],
       auto simp add: nat_into_M)
apply (rule_tac env="[B,A,n,Memrel(eclose({n}))]" in DPow_LsetI)
apply (rule sep_rules satisfies_MH_iff_sats is_wfrec_iff_sats | simp)+
done
```

13.4.6 The Lambda Replacement Case

```
lemma formula_rec_lambda_replacement_Reflects:
 "REFLECTS [λx. ∃u[L]. u ∈ B &
     mem_formula(L,u) &
     (∃c[L].
         is_formula_case
          (L, satisfies_is_a(L,A), satisfies_is_b(L,A),
           satisfies_is_c(L,A,g), satisfies_is_d(L,A,g),
           u, c) &
         pair(L,u,c,x)),
  λi x. ∃u ∈ Lset(i). u ∈ B & mem_formula(##Lset(i),u) &
     (∃c ∈ Lset(i).
         is_formula_case
          (##Lset(i), satisfies_is_a(##Lset(i),A), satisfies_is_b(##Lset(i),A),
           satisfies_is_c(##Lset(i),A,g), satisfies_is_d(##Lset(i),A,g),
           u, c) &
         pair(##Lset(i),u,c,x))]"
by (intro FOL_reflections function_reflections mem_formula_reflection
          is_formula_case_reflection satisfies_is_a_reflection
          satisfies_is_b_reflection satisfies_is_c_reflection
          satisfies_is_d_reflection)
```

```
lemma formula_rec_lambda_replacement:
```

— For the *transrec*

```
  "[|L(g); L(A)|] ==>
    strong_replacement (L,
       λx y. mem_formula(L,x) &
             (∃c[L]. is_formula_case(L, satisfies_is_a(L,A),
                                 satisfies_is_b(L,A),
                                 satisfies_is_c(L,A,g),
                                 satisfies_is_d(L,A,g), x, c) &
             pair(L, x, c, y)))"
apply (rule strong_replacementI)
apply (rule_tac u="{B,A,g}"
        in gen_separation_multi [OF formula_rec_lambda_replacement_Reflects],

       auto)
```

```
apply (rule_tac env="[A,g,B]" in DPow_LsetI)
apply (rule sep_rules mem_formula_iff_sats
          formula_case_iff_sats satisfies_is_a_iff_sats
          satisfies_is_b_iff_sats satisfies_is_c_iff_sats
          satisfies_is_d_iff_sats | simp)+
done
```

13.5 Instantiating *M_satisfies*

```
lemma M_satisfies_axioms_L: "M_satisfies_axioms(L)"
  apply (rule M_satisfies_axioms.intro)
       apply (assumption | rule
         Member_replacement Equal_replacement
         Nand_replacement Forall_replacement
         formula_rec_replacement formula_rec_lambda_replacement)+
  done

theorem M_satisfies_L: "PROP M_satisfies(L)"
  apply (rule M_satisfies.intro)
   apply (rule M_eclose_L)
  apply (rule M_satisfies_axioms_L)
  done
```

Finally: the point of the whole theory!

```
lemmas satisfies_closed = M_satisfies.satisfies_closed [OF M_satisfies_L]
   and satisfies_abs = M_satisfies.satisfies_abs [OF M_satisfies_L]

end
```

14 Absoluteness for the Definable Powerset Function

```
theory DPow_absolute imports Satisfies_absolute begin
```

14.1 Preliminary Internalizations

14.1.1 The Operator *is_formula_rec*

The three arguments of *p* are always 2, 1, 0. It is buried within 11 quantifiers!!

```
definition
  formula_rec_fm :: "[i, i, i]=>i" where
 "formula_rec_fm(mh,p,z) ==
    Exists(Exists(Exists(
      And(finite_ordinal_fm(2),
       And(depth_fm(p#+3,2),
        And(succ_fm(2,1),
```

```
          And(fun_apply_fm(0,p#+3,z#+3), is_transrec_fm(mh,1,0))))))))"
```

lemma *is_formula_rec_type [TC]:*
```
     "[| p ∈ formula; x ∈ nat; z ∈ nat |]
      ==> formula_rec_fm(p,x,z) ∈ formula"
```
by *(simp add: formula_rec_fm_def)*

lemma *sats_formula_rec_fm:*
  assumes *MH_iff_sats:*
```
      "!!a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10.
        [|a0∈A; a1∈A; a2∈A; a3∈A; a4∈A; a5∈A; a6∈A; a7∈A; a8∈A; a9∈A;
a10∈A|]
        ==> MH(a2, a1, a0) <->
            sats(A, p, Cons(a0,Cons(a1,Cons(a2,Cons(a3,
                          Cons(a4,Cons(a5,Cons(a6,Cons(a7,
                                  Cons(a8,Cons(a9,Cons(a10,env)))))))))))"
```
  shows
```
      "[|x ∈ nat; z ∈ nat; env ∈ list(A)|]
       ==> sats(A, formula_rec_fm(p,x,z), env) <->
           is_formula_rec(##A, MH, nth(x,env), nth(z,env))"
```
by *(simp add: formula_rec_fm_def sats_is_transrec_fm is_formula_rec_def*
```
              MH_iff_sats [THEN iff_sym])
```

lemma *formula_rec_iff_sats:*
  assumes *MH_iff_sats:*
```
      "!!a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10.
        [|a0∈A; a1∈A; a2∈A; a3∈A; a4∈A; a5∈A; a6∈A; a7∈A; a8∈A; a9∈A;
a10∈A|]
        ==> MH(a2, a1, a0) <->
            sats(A, p, Cons(a0,Cons(a1,Cons(a2,Cons(a3,
                          Cons(a4,Cons(a5,Cons(a6,Cons(a7,
                                  Cons(a8,Cons(a9,Cons(a10,env)))))))))))"
```
  shows
```
  "[|nth(i,env) = x; nth(k,env) = z;
      i ∈ nat; k ∈ nat; env ∈ list(A)|]
   ==> is_formula_rec(##A, MH, x, z) <-> sats(A, formula_rec_fm(p,i,k),
env)"
```
by *(simp add: sats_formula_rec_fm [OF MH_iff_sats])*

theorem *formula_rec_reflection:*
  assumes *MH_reflection:*
```
    "!!f' f g h. REFLECTS[λx. MH(L, f'(x), f(x), g(x), h(x)),
                     λi x. MH(##Lset(i), f'(x), f(x), g(x), h(x))]"
```
  shows *"REFLECTS[λx. is_formula_rec(L, MH(L,x), f(x), h(x)),*
```
               λi x. is_formula_rec(##Lset(i), MH(##Lset(i),x), f(x),
h(x))]"
```
apply *(simp (no_asm_use) only: is_formula_rec_def)*
apply *(intro FOL_reflections function_reflections fun_plus_reflections*

```
            depth_reflection is_transrec_reflection MH_reflection)
done
```

14.1.2 The Operator *is_satisfies*

```
definition
  satisfies_fm :: "[i,i,i]=>i" where
    "satisfies_fm(x) == formula_rec_fm (satisfies_MH_fm(x#+5#+6, 2, 1,
0))"

lemma is_satisfies_type [TC]:
     "[| x ∈ nat; y ∈ nat; z ∈ nat |] ==> satisfies_fm(x,y,z) ∈ formula"
by (simp add: satisfies_fm_def)

lemma sats_satisfies_fm [simp]:
   "[| x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, satisfies_fm(x,y,z), env) <->
        is_satisfies(##A, nth(x,env), nth(y,env), nth(z,env))"
by (simp add: satisfies_fm_def is_satisfies_def sats_satisfies_MH_fm
              sats_formula_rec_fm)

lemma satisfies_iff_sats:
      "[| nth(i,env) = x; nth(j,env) = y; nth(k,env) = z;
          i ∈ nat; j ∈ nat; k ∈ nat; env ∈ list(A)|]
       ==> is_satisfies(##A, x, y, z) <-> sats(A, satisfies_fm(i,j,k),
env)"
by (simp add: sats_satisfies_fm)

theorem satisfies_reflection:
     "REFLECTS[λx. is_satisfies(L,f(x),g(x),h(x)),
               λi x. is_satisfies(##Lset(i),f(x),g(x),h(x))]"
apply (simp only: is_satisfies_def)
apply (intro formula_rec_reflection satisfies_MH_reflection)
done
```

14.2 Relativization of the Operator *DPow'*

```
lemma DPow'_eq:
  "DPow'(A) = {z . ep ∈ list(A) * formula,
                   ∃env ∈ list(A). ∃p ∈ formula.
                      ep = <env,p> & z = {x∈A. sats(A, p, Cons(x,env))}}"
by (simp add: DPow'_def, blast)
```

Relativize the use of *λA p env. sats(A, p, env)* within *DPow'* (the comprehension).

```
definition
  is_DPow_sats :: "[i=>o,i,i,i,i] => o" where
   "is_DPow_sats(M,A,env,p,x) ==
     ∀n1[M]. ∀e[M]. ∀sp[M].
```

```
            is_satisfies(M,A,p,sp) --> is_Cons(M,x,env,e) -->
            fun_apply(M, sp, e, n1) --> number1(M, n1)"
```

lemma *(in M_satisfies) DPow_sats_abs:*
```
    "[| M(A); env ∈ list(A); p ∈ formula; M(x) |]
    ==> is_DPow_sats(M,A,env,p,x) <-> sats(A, p, Cons(x,env))"
```
apply *(subgoal_tac "M(env)")*
 apply *(simp add: is_DPow_sats_def satisfies_closed satisfies_abs)*
apply *(blast dest: transM)*
done

lemma *(in M_satisfies) Collect_DPow_sats_abs:*
```
    "[| M(A); env ∈ list(A); p ∈ formula |]
    ==> Collect(A, is_DPow_sats(M,A,env,p)) =
        {x ∈ A. sats(A, p, Cons(x,env))}"
```
by *(simp add: DPow_sats_abs transM [of _ A])*

14.2.1 The Operator *is_DPow_sats*, Internalized

definition
```
  DPow_sats_fm :: "[i,i,i,i]=>i" where
  "DPow_sats_fm(A,env,p,x) ==
   Forall(Forall(Forall(
     Implies(satisfies_fm(A#+3,p#+3,0),
       Implies(Cons_fm(x#+3,env#+3,1),
         Implies(fun_apply_fm(0,1,2), number1_fm(2)))))))"
```

lemma *is_DPow_sats_type [TC]:*
```
     "[| A ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat |]
      ==> DPow_sats_fm(A,x,y,z) ∈ formula"
```
by *(simp add: DPow_sats_fm_def)*

lemma *sats_DPow_sats_fm [simp]:*
```
   "[| u ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
    ==> sats(A, DPow_sats_fm(u,x,y,z), env) <->
        is_DPow_sats(##A, nth(u,env), nth(x,env), nth(y,env), nth(z,env))"
```
by *(simp add: DPow_sats_fm_def is_DPow_sats_def)*

lemma *DPow_sats_iff_sats:*
```
  "[| nth(u,env) = nu; nth(x,env) = nx; nth(y,env) = ny; nth(z,env) =
nz;
      u ∈ nat; x ∈ nat; y ∈ nat; z ∈ nat; env ∈ list(A)|]
   ==> is_DPow_sats(##A,nu,nx,ny,nz) <->
       sats(A, DPow_sats_fm(u,x,y,z), env)"
```
by *simp*

theorem *DPow_sats_reflection:*
```
     "REFLECTS[λx. is_DPow_sats(L,f(x),g(x),h(x),g'(x)),
               λi x. is_DPow_sats(##Lset(i),f(x),g(x),h(x),g'(x))]"
```

```
apply (unfold is_DPow_sats_def)
apply (intro FOL_reflections function_reflections extra_reflections
             satisfies_reflection)
done
```

14.3 A Locale for Relativizing the Operator *DPow'*

```
locale M_DPow = M_satisfies +
 assumes sep:
   "[| M(A); env ∈ list(A); p ∈ formula |]
    ==> separation(M, λx. is_DPow_sats(M,A,env,p,x))"
 and rep:
    "M(A)
    ==> strong_replacement (M,
         λep z. ∃env[M]. ∃p[M]. mem_formula(M,p) & mem_list(M,A,env)
&
                  pair(M,env,p,ep) &
                  is_Collect(M, A, λx. is_DPow_sats(M,A,env,p,x), z))"

lemma (in M_DPow) sep':
   "[| M(A); env ∈ list(A); p ∈ formula |]
    ==> separation(M, λx. sats(A, p, Cons(x,env)))"
by (insert sep [of A env p], simp add: DPow_sats_abs)

lemma (in M_DPow) rep':
   "M(A)
    ==> strong_replacement (M,
         λep z. ∃env∈list(A). ∃p∈formula.
                  ep = <env,p> & z = {x ∈ A . sats(A, p, Cons(x, env))})"

by (insert rep [of A], simp add: Collect_DPow_sats_abs)


lemma univalent_pair_eq:
     "univalent (M, A, λxy z. ∃x∈B. ∃y∈C. xy = ⟨x,y⟩ ∧ z = f(x,y))"
by (simp add: univalent_def, blast)

lemma (in M_DPow) DPow'_closed: "M(A) ==> M(DPow'(A))"
apply (simp add: DPow'_eq)
apply (fast intro: rep' sep' univalent_pair_eq)
done
```

Relativization of the Operator *DPow'*

```
definition
  is_DPow' :: "[i=>o,i,i] => o" where
    "is_DPow'(M,A,Z) ==
       ∀X[M]. X ∈ Z <->
         subset(M,X,A) &
           (∃env[M]. ∃p[M]. mem_formula(M,p) & mem_list(M,A,env) &
```

```
                  is_Collect(M, A, is_DPow_sats(M,A,env,p), X))"
```

lemma *(in M_DPow) DPow'_abs:*

```
    "[|M(A); M(Z)|] ==> is_DPow'(M,A,Z) <-> Z = DPow'(A)"
apply (rule iffI)
 prefer 2 apply (simp add: is_DPow'_def DPow'_def Collect_DPow_sats_abs)

apply (rule M_equalityI)
apply (simp add: is_DPow'_def DPow'_def Collect_DPow_sats_abs, assumption)
apply (erule DPow'_closed)
done
```

14.4 Instantiating the Locale *M_DPow*

14.4.1 The Instance of Separation

```
lemma DPow_separation:
    "[| L(A); env ∈ list(A); p ∈ formula |]
     ==> separation(L, λx. is_DPow_sats(L,A,env,p,x))"
apply (rule gen_separation_multi [OF DPow_sats_reflection, of "{A,env,p}"],

       auto intro: transL)
apply (rule_tac env="[A,env,p]" in DPow_LsetI)
apply (rule DPow_sats_iff_sats sep_rules | simp)+
done
```

14.4.2 The Instance of Replacement

```
lemma DPow_replacement_Reflects:
 "REFLECTS [λx. ∃u[L]. u ∈ B &
             (∃env[L]. ∃p[L].
               mem_formula(L,p) & mem_list(L,A,env) & pair(L,env,p,u)
&
               is_Collect (L, A, is_DPow_sats(L,A,env,p), x)),
    λi x. ∃u ∈ Lset(i). u ∈ B &
             (∃env ∈ Lset(i). ∃p ∈ Lset(i).
               mem_formula(##Lset(i),p) & mem_list(##Lset(i),A,env) &

               pair(##Lset(i),env,p,u) &
               is_Collect (##Lset(i), A, is_DPow_sats(##Lset(i),A,env,p),
x))]"
apply (unfold is_Collect_def)
apply (intro FOL_reflections function_reflections mem_formula_reflection
          mem_list_reflection DPow_sats_reflection)
done

lemma DPow_replacement:
    "L(A)
    ==> strong_replacement (L,
```

```
         λep z. ∃env[L]. ∃p[L]. mem_formula(L,p) & mem_list(L,A,env)
&
                  pair(L,env,p,ep) &
                  is_Collect(L, A, λx. is_DPow_sats(L,A,env,p,x), z))"
apply (rule strong_replacementI)
apply (rule_tac u="{A,B}"
         in gen_separation_multi [OF DPow_replacement_Reflects],
       auto)
apply (unfold is_Collect_def)
apply (rule_tac env="[A,B]" in DPow_LsetI)
apply (rule sep_rules mem_formula_iff_sats mem_list_iff_sats
            DPow_sats_iff_sats | simp)+
done
```

14.4.3 Actually Instantiating the Locale

```
lemma M_DPow_axioms_L: "M_DPow_axioms(L)"
  apply (rule M_DPow_axioms.intro)
   apply (assumption | rule DPow_separation DPow_replacement)+
  done

theorem M_DPow_L: "PROP M_DPow(L)"
  apply (rule M_DPow.intro)
   apply (rule M_satisfies_L)
  apply (rule M_DPow_axioms_L)
  done

lemmas DPow'_closed [intro, simp] = M_DPow.DPow'_closed [OF M_DPow_L]
  and DPow'_abs [intro, simp] = M_DPow.DPow'_abs [OF M_DPow_L]
```

14.4.4 The Operator *is_Collect*

The formula *is_P* has one free variable, 0, and it is enclosed within a single quantifier.

```
definition
  Collect_fm :: "[i, i, i]=>i" where
 "Collect_fm(A,is_P,z) ==
        Forall(Iff(Member(0,succ(z)),
                   And(Member(0,succ(A)), is_P)))"

lemma is_Collect_type [TC]:
     "[| is_P ∈ formula; x ∈ nat; y ∈ nat |]
      ==> Collect_fm(x,is_P,y) ∈ formula"
by (simp add: Collect_fm_def)

lemma sats_Collect_fm:
  assumes is_P_iff_sats:
      "!!a. a ∈ A ==> is_P(a) <-> sats(A, p, Cons(a, env))"
  shows
```

```
      "[|x ∈ nat; y ∈ nat; env ∈ list(A)|]
       ==> sats(A, Collect_fm(x,p,y), env) <->
           is_Collect(##A, nth(x,env), is_P, nth(y,env))"
by (simp add: Collect_fm_def is_Collect_def is_P_iff_sats [THEN iff_sym])

lemma Collect_iff_sats:
  assumes is_P_iff_sats:
      "!!a. a ∈ A ==> is_P(a) <-> sats(A, p, Cons(a, env))"
  shows
  "[| nth(i,env) = x; nth(j,env) = y;
      i ∈ nat; j ∈ nat; env ∈ list(A)|]
   ==> is_Collect(##A, x, is_P, y) <-> sats(A, Collect_fm(i,p,j), env)"
by (simp add: sats_Collect_fm [OF is_P_iff_sats])
```

The second argument of `is_P` gives it direct access to `x`, which is essential for handling free variable references.

```
theorem Collect_reflection:
  assumes is_P_reflection:
    "!!h f g. REFLECTS[λx. is_P(L, f(x), g(x)),
                     λi x. is_P(##Lset(i), f(x), g(x))]"
  shows "REFLECTS[λx. is_Collect(L, f(x), is_P(L,x), g(x)),
               λi x. is_Collect(##Lset(i), f(x), is_P(##Lset(i), x), g(x))]"
apply (simp (no_asm_use) only: is_Collect_def)
apply (intro FOL_reflections is_P_reflection)
done
```

14.4.5 The Operator `is_Replace`

BEWARE! The formula `is_P` has free variables 0, 1 and not the usual 1, 0! It is enclosed within two quantifiers.

```
definition
  Replace_fm :: "[i, i, i]=>i" where
  "Replace_fm(A,is_P,z) ==
        Forall(Iff(Member(0,succ(z)),
                   Exists(And(Member(0,A#+2), is_P))))"

lemma is_Replace_type [TC]:
     "[| is_P ∈ formula; x ∈ nat; y ∈ nat |]
      ==> Replace_fm(x,is_P,y) ∈ formula"
by (simp add: Replace_fm_def)

lemma sats_Replace_fm:
  assumes is_P_iff_sats:
      "!!a b. [|a ∈ A; b ∈ A|]
              ==> is_P(a,b) <-> sats(A, p, Cons(a,Cons(b,env)))"
  shows
      "[|x ∈ nat; y ∈ nat; env ∈ list(A)|]
       ==> sats(A, Replace_fm(x,p,y), env) <->
```

```
          is_Replace(##A, nth(x,env), is_P, nth(y,env))"
by (simp add: Replace_fm_def is_Replace_def is_P_iff_sats [THEN iff_sym])

lemma Replace_iff_sats:
  assumes is_P_iff_sats:
      "!!a b. [|a ∈ A; b ∈ A|]
              ==> is_P(a,b) <-> sats(A, p, Cons(a,Cons(b,env)))"
  shows
  "[| nth(i,env) = x; nth(j,env) = y;
      i ∈ nat; j ∈ nat; env ∈ list(A)|]
   ==> is_Replace(##A, x, is_P, y) <-> sats(A, Replace_fm(i,p,j), env)"
by (simp add: sats_Replace_fm [OF is_P_iff_sats])
```

The second argument of `is_P` gives it direct access to `x`, which is essential for handling free variable references.

```
theorem Replace_reflection:
  assumes is_P_reflection:
    "!!h f g. REFLECTS[λx. is_P(L, f(x), g(x), h(x)),
                     λi x. is_P(##Lset(i), f(x), g(x), h(x))]"
  shows "REFLECTS[λx. is_Replace(L, f(x), is_P(L,x), g(x)),
              λi x. is_Replace(##Lset(i), f(x), is_P(##Lset(i), x), g(x))]"
apply (simp (no_asm_use) only: is_Replace_def)
apply (intro FOL_reflections is_P_reflection)
done
```

14.4.6 The Operator `is_DPow'`, Internalized

```
definition
  DPow'_fm :: "[i,i]=>i" where
    "DPow'_fm(A,Z) ==
      Forall(
       Iff(Member(0,succ(Z)),
        And(subset_fm(0,succ(A)),
         Exists(Exists(
          And(mem_formula_fm(0),
           And(mem_list_fm(A#+3,1),
            Collect_fm(A#+3,
                      DPow_sats_fm(A#+4, 2, 1, 0), 2))))))))"

lemma is_DPow'_type [TC]:
     "[| x ∈ nat; y ∈ nat |] ==> DPow'_fm(x,y) ∈ formula"
by (simp add: DPow'_fm_def)

lemma sats_DPow'_fm [simp]:
   "[| x ∈ nat; y ∈ nat; env ∈ list(A)|]
    ==> sats(A, DPow'_fm(x,y), env) <->
        is_DPow'(##A, nth(x,env), nth(y,env))"
by (simp add: DPow'_fm_def is_DPow'_def sats_subset_fm' sats_Collect_fm)
```

```
lemma DPow'_iff_sats:
      "[| nth(i,env) = x; nth(j,env) = y;
          i ∈ nat; j ∈ nat; env ∈ list(A)|]
       ==> is_DPow'(##A, x, y) <-> sats(A, DPow'_fm(i,j), env)"
by (simp add: sats_DPow'_fm)

theorem DPow'_reflection:
     "REFLECTS[λx. is_DPow'(L,f(x),g(x)),
               λi x. is_DPow'(##Lset(i),f(x),g(x))]"
apply (simp only: is_DPow'_def)
apply (intro FOL_reflections function_reflections mem_formula_reflection
             mem_list_reflection Collect_reflection DPow_sats_reflection)
done
```

14.5 A Locale for Relativizing the Operator *Lset*

```
definition
  transrec_body :: "[i=>o,i,i,i,i] => o" where
    "transrec_body(M,g,x) ==
      λy z. ∃gy[M]. y ∈ x & fun_apply(M,g,y,gy) & is_DPow'(M,gy,z)"

lemma (in M_DPow) transrec_body_abs:
     "[|M(x); M(g); M(z)|]
    ==> transrec_body(M,g,x,y,z) <-> y ∈ x & z = DPow'(g`y)"
by (simp add: transrec_body_def DPow'_abs transM [of _ x])

locale M_Lset = M_DPow +
 assumes strong_rep:
   "[|M(x); M(g)|] ==> strong_replacement(M, λy z. transrec_body(M,g,x,y,z))"
 and transrec_rep:
    "M(i) ==> transrec_replacement(M, λx f u.
              ∃r[M]. is_Replace(M, x, transrec_body(M,f,x), r) &
                     big_union(M, r, u), i)"


lemma (in M_Lset) strong_rep':
   "[|M(x); M(g)|]
    ==> strong_replacement(M, λy z. y ∈ x & z = DPow'(g`y))"
by (insert strong_rep [of x g], simp add: transrec_body_abs)

lemma (in M_Lset) DPow_apply_closed:
   "[|M(f); M(x); y∈x|] ==> M(DPow'(f`y))"
by (blast intro: DPow'_closed dest: transM)

lemma (in M_Lset) RepFun_DPow_apply_closed:
   "[|M(f); M(x)|] ==> M({DPow'(f`y). y∈x})"
by (blast intro: DPow_apply_closed RepFun_closed2 strong_rep')

lemma (in M_Lset) RepFun_DPow_abs:
```

"[|M(x); M(f); M(r) |]
==> is_Replace(M, x, λy z. transrec_body(M,f,x,y,z), r) <->
r = {DPow'(f`y). y∈x}"
apply *(simp add: transrec_body_abs RepFun_def)*
apply *(rule iff_trans)*
apply *(rule Replace_abs)*
apply *(simp_all add: DPow_apply_closed strong_rep')*
done

lemma *(in M_Lset) transrec_rep':*
"M(i) ==> transrec_replacement(M, λx f u. u = (⋃y∈x. DPow'(f ` y)), i)"
apply *(insert transrec_rep [of i])*
apply *(simp add: RepFun_DPow_apply_closed RepFun_DPow_abs transrec_replacement_def)*
done

Relativization of the Operator `Lset`

definition
is_Lset :: "[i=>o, i, i] => o" **where**
— We can use the term language below because `is_Lset` will not have to be internalized: it isn't used in any instance of separation.
"is_Lset(M,a,z) == is_transrec(M, %x f u. u = (⋃y∈x. DPow'(f`y)), a, z)"

lemma *(in M_Lset) Lset_abs:*
"[|Ord(i); M(i); M(z)|]
==> is_Lset(M,i,z) <-> z = Lset(i)"
apply *(simp add: is_Lset_def Lset_eq_transrec_DPow')*
apply *(rule transrec_abs)*
apply *(simp_all add: transrec_rep' relation2_def RepFun_DPow_apply_closed)*
done

lemma *(in M_Lset) Lset_closed:*
"[|Ord(i); M(i)|] ==> M(Lset(i))"
apply *(simp add: Lset_eq_transrec_DPow')*
apply *(rule transrec_closed [OF transrec_rep'])*
apply *(simp_all add: relation2_def RepFun_DPow_apply_closed)*
done

14.6 Instantiating the Locale `M_Lset`

14.6.1 The First Instance of Replacement

lemma *strong_rep_Reflects:*
"REFLECTS [λu. ∃v[L]. v ∈ B & (∃gy[L].
v ∈ x & fun_apply(L,g,v,gy) & is_DPow'(L,gy,u)),
λi u. ∃v ∈ Lset(i). v ∈ B & (∃gy ∈ Lset(i).
v ∈ x & fun_apply(##Lset(i),g,v,gy) & is_DPow'(##Lset(i),gy,u))]"
by *(intro FOL_reflections function_reflections DPow'_reflection)*

```
lemma strong_rep:
   "[|L(x); L(g)|] ==> strong_replacement(L, λy z. transrec_body(L,g,x,y,z))"
apply (unfold transrec_body_def)
apply (rule strong_replacementI)
apply (rule_tac u="{x,g,B}"
         in gen_separation_multi [OF strong_rep_Reflects], auto)
apply (rule_tac env="[x,g,B]" in DPow_LsetI)
apply (rule sep_rules DPow'_iff_sats | simp)+
done
```

14.6.2 The Second Instance of Replacement

```
lemma transrec_rep_Reflects:
 "REFLECTS [λx. ∃v[L]. v ∈ B &
              (∃y[L]. pair(L,v,y,x) &
             is_wfrec (L, λx f u. ∃r[L].
                is_Replace (L, x, λy z.
                     ∃gy[L]. y ∈ x & fun_apply(L,f,y,gy) &
                      is_DPow'(L,gy,z), r) & big_union(L,r,u), mr, v,
y)),
    λi x. ∃v ∈ Lset(i). v ∈ B &
              (∃y ∈ Lset(i). pair(##Lset(i),v,y,x) &
             is_wfrec (##Lset(i), λx f u. ∃r ∈ Lset(i).
                is_Replace (##Lset(i), x, λy z.
                     ∃gy ∈ Lset(i). y ∈ x & fun_apply(##Lset(i),f,y,gy)
&
                      is_DPow'(##Lset(i),gy,z), r) &
                      big_union(##Lset(i),r,u), mr, v, y))]"
apply (simp only: rex_setclass_is_bex [symmetric])
  — Convert ∃y∈Lset(i) to ∃y[##Lset(i)] within the body of the is_wfrec
application.
apply (intro FOL_reflections function_reflections
          is_wfrec_reflection Replace_reflection DPow'_reflection)
done


lemma transrec_rep:
    "[|L(j)|]
    ==> transrec_replacement(L, λx f u.
              ∃r[L]. is_Replace(L, x, transrec_body(L,f,x), r) &
                     big_union(L, r, u), j)"
apply (rule transrec_replacementI, assumption)
apply (unfold transrec_body_def)
apply (rule strong_replacementI)
apply (rule_tac u="{j,B,Memrel(eclose({j}))}"
         in gen_separation_multi [OF transrec_rep_Reflects], auto)
apply (rule_tac env="[j,B,Memrel(eclose({j}))]" in DPow_LsetI)
apply (rule sep_rules is_wfrec_iff_sats Replace_iff_sats DPow'_iff_sats
```

```
|
       simp)+
done
```

14.6.3 Actually Instantiating *M_Lset*

```
lemma M_Lset_axioms_L: "M_Lset_axioms(L)"
  apply (rule M_Lset_axioms.intro)
       apply (assumption | rule strong_rep transrec_rep)+
  done

theorem M_Lset_L: "PROP M_Lset(L)"
  apply (rule M_Lset.intro)
   apply (rule M_DPow_L)
  apply (rule M_Lset_axioms_L)
  done
```

Finally: the point of the whole theory!

```
lemmas Lset_closed = M_Lset.Lset_closed [OF M_Lset_L]
   and Lset_abs = M_Lset.Lset_abs [OF M_Lset_L]
```

14.7 The Notion of Constructible Set

```
definition
  constructible :: "[i=>o,i] => o" where
    "constructible(M,x) ==
       ∃i[M]. ∃Li[M]. ordinal(M,i) & is_Lset(M,i,Li) & x ∈ Li"

theorem V_equals_L_in_L:
  "L(x) ==> constructible(L,x)"
apply (simp add: constructible_def Lset_abs Lset_closed)
apply (simp add: L_def)
apply (blast intro: Ord_in_L)
done

end
```

15 The Axiom of Choice Holds in L!

```
theory AC_in_L imports Formula begin
```

15.1 Extending a Wellordering over a List – Lexicographic Power

This could be moved into a library.

```
consts
  rlist   :: "[i,i]=>i"
```

```
inductive
  domains "rlist(A,r)" ⊆ "list(A) * list(A)"
  intros
    shorterI:
      "[| length(l') < length(l); l' ∈ list(A); l ∈ list(A) |]
       ==> <l', l> ∈ rlist(A,r)"

    sameI:
      "[| <l',l> ∈ rlist(A,r); a ∈ A |]
       ==> <Cons(a,l'), Cons(a,l)> ∈ rlist(A,r)"

    diffI:
      "[| length(l') = length(l); <a',a> ∈ r;
          l' ∈ list(A); l ∈ list(A); a' ∈ A; a ∈ A |]
       ==> <Cons(a',l'), Cons(a,l)> ∈ rlist(A,r)"
  type_intros list.intros
```

15.1.1 Type checking

```
lemmas rlist_type = rlist.dom_subset

lemmas field_rlist = rlist_type [THEN field_rel_subset]
```

15.1.2 Linearity

```
lemma rlist_Nil_Cons [intro]:
    "[|a ∈ A; l ∈ list(A)|] ==> <[], Cons(a,l)> ∈ rlist(A, r)"
by (simp add: shorterI)

lemma linear_rlist:
    "linear(A,r) ==> linear(list(A),rlist(A,r))"
apply (simp (no_asm_simp) add: linear_def)
apply (rule ballI)
apply (induct_tac x)
 apply (rule ballI)
 apply (induct_tac y)
  apply (simp_all add: shorterI)
apply (rule ballI)
apply (erule_tac a=y in list.cases)
 apply (rename_tac [2] a2 l2)
 apply (rule_tac [2] i = "length(l)" and j = "length(l2)" in Ord_linear_lt)
     apply (simp_all add: shorterI)
apply (erule_tac x=a and y=a2 in linearE)
    apply (simp_all add: diffI)
apply (blast intro: sameI)
done
```

15.1.3 Well-foundedness

Nothing preceeds Nil in this ordering.

```
inductive_cases rlist_NilE: " <l,[]> ∈ rlist(A,r)"

inductive_cases rlist_ConsE: " <l', Cons(x,l)> ∈ rlist(A,r)"

lemma not_rlist_Nil [simp]: " <l,[]> ∉ rlist(A,r)"
by (blast intro: elim: rlist_NilE)

lemma rlist_imp_length_le: "<l',l> ∈ rlist(A,r) ==> length(l') ≤ length(l)"
apply (erule rlist.induct)
apply (simp_all add: leI)
done

lemma wf_on_rlist_n:
  "[| n ∈ nat; wf[A](r) |] ==> wf[{l ∈ list(A). length(l) = n}](rlist(A,r))"
apply (induct_tac n)
 apply (rule wf_onI2, simp)
apply (rule wf_onI2, clarify)
apply (erule_tac a=y in list.cases, clarify)
 apply (simp (no_asm_use))
apply clarify
apply (simp (no_asm_use))
apply (subgoal_tac "∀ l2 ∈ list(A). length(l2) = x --> Cons(a,l2) ∈ B",
blast)
apply (erule_tac a=a in wf_on_induct, assumption)
apply (rule ballI)
apply (rule impI)
apply (erule_tac a=l2 in wf_on_induct, blast, clarify)
apply (rename_tac a' l2 l')
apply (drule_tac x="Cons(a',l')" in bspec, typecheck)
apply simp
apply (erule mp, clarify)
apply (erule rlist_ConsE, auto)
done

lemma list_eq_UN_length: "list(A) = (⋃n∈nat. {l ∈ list(A). length(l)
= n})"
by (blast intro: length_type)


lemma wf_on_rlist: "wf[A](r) ==> wf[list(A)](rlist(A,r))"
apply (subst list_eq_UN_length)
apply (rule wf_on_Union)
  apply (rule wf_imp_wf_on [OF wf_Memrel [of nat]])
 apply (simp add: wf_on_rlist_n)
apply (frule rlist_type [THEN subsetD])
apply (simp add: length_type)
```

```
apply (drule rlist_imp_length_le)
apply (erule leE)
apply (simp_all add: lt_def)
done
```

```
lemma wf_rlist: "wf(r) ==> wf(rlist(field(r),r))"
apply (simp add: wf_iff_wf_on_field)
apply (rule wf_on_subset_A [OF _ field_rlist])
apply (blast intro: wf_on_rlist)
done
```

```
lemma well_ord_rlist:
     "well_ord(A,r) ==> well_ord(list(A), rlist(A,r))"
apply (rule well_ordI)
apply (simp add: well_ord_def wf_on_rlist)
apply (simp add: well_ord_def tot_ord_def linear_rlist)
done
```

15.2 An Injection from Formulas into the Natural Numbers

There is a well-known bijection between *nat* × *nat* and *nat* given by the expression f(m,n) = triangle(m+n) + m, where triangle(k) enumerates the triangular numbers and can be defined by triangle(0)=0, triangle(succ(k)) = succ(k + triangle(k)). Some small amount of effort is needed to show that f is a bijection. We already know that such a bijection exists by the theorem *well_ord_InfCard_square_eq*:

$[\![\mathit{well\_ord}(A,\ r);\ \mathit{InfCard}(|A|)]\!] \Longrightarrow A \times A \approx A$

However, this result merely states that there is a bijection between the two sets. It provides no means of naming a specific bijection. Therefore, we conduct the proofs under the assumption that a bijection exists. The simplest way to organize this is to use a locale.

Locale for any arbitrary injection between *nat* × *nat* and *nat*

```
locale Nat_Times_Nat =
  fixes fn
  assumes fn_inj: "fn ∈ inj(nat*nat, nat)"
```

```
consts   enum :: "[i,i]=>i"
primrec
  "enum(f, Member(x,y)) = f ‘ <0, f ‘ <x,y>>"
  "enum(f, Equal(x,y)) = f ‘ <1, f ‘ <x,y>>"
  "enum(f, Nand(p,q)) = f ‘ <2, f ‘ <enum(f,p), enum(f,q)>>"
  "enum(f, Forall(p)) = f ‘ <succ(2), enum(f,p)>"
```

```
lemma (in Nat_Times_Nat) fn_type [TC,simp]:
    "[|x ∈ nat; y ∈ nat|] ==> fn`<x,y> ∈ nat"
by (blast intro: inj_is_fun [OF fn_inj] apply_funtype)

lemma (in Nat_Times_Nat) fn_iff:
    "[|x ∈ nat; y ∈ nat; u ∈ nat; v ∈ nat|]
     ==> (fn`<x,y> = fn`<u,v>) <-> (x=u & y=v)"
by (blast dest: inj_apply_equality [OF fn_inj])

lemma (in Nat_Times_Nat) enum_type [TC,simp]:
    "p ∈ formula ==> enum(fn,p) ∈ nat"
by (induct_tac p, simp_all)

lemma (in Nat_Times_Nat) enum_inject [rule_format]:
    "p ∈ formula ==> ∀q∈formula. enum(fn,p) = enum(fn,q) --> p=q"
apply (induct_tac p, simp_all)
   apply (rule ballI)
   apply (erule formula.cases)
   apply (simp_all add: fn_iff)
  apply (rule ballI)
  apply (erule formula.cases)
  apply (simp_all add: fn_iff)
 apply (rule ballI)
 apply (erule_tac a=qa in formula.cases)
 apply (simp_all add: fn_iff)
 apply blast
apply (rule ballI)
apply (erule_tac a=q in formula.cases)
apply (simp_all add: fn_iff, blast)
done

lemma (in Nat_Times_Nat) inj_formula_nat:
    "(λp ∈ formula. enum(fn,p)) ∈ inj(formula, nat)"
apply (simp add: inj_def lam_type)
apply (blast intro: enum_inject)
done

lemma (in Nat_Times_Nat) well_ord_formula:
    "well_ord(formula, measure(formula, enum(fn)))"
apply (rule well_ord_measure, simp)
apply (blast intro: enum_inject)
done

lemmas nat_times_nat_lepoll_nat =
    InfCard_nat [THEN InfCard_square_eqpoll, THEN eqpoll_imp_lepoll]
```

Not needed–but interesting?

```
theorem formula_lepoll_nat: "formula ≲ nat"
apply (insert nat_times_nat_lepoll_nat)
```

apply *(unfold lepoll_def)*
apply *(blast intro: Nat_Times_Nat.inj_formula_nat Nat_Times_Nat.intro)*
done

15.3 Defining the Wellordering on *DPow(A)*

The objective is to build a wellordering on *DPow(A)* from a given one on *A*. We first introduce wellorderings for environments, which are lists built over *A*. We combine it with the enumeration of formulas. The order type of the resulting wellordering gives us a map from (environment, formula) pairs into the ordinals. For each member of *DPow(A)*, we take the minimum such ordinal.

definition
```
  env_form_r :: "[i,i,i]=>i" where
    — wellordering on (environment, formula) pairs
   "env_form_r(f,r,A) ==
      rmult(list(A), rlist(A, r),
            formula, measure(formula, enum(f)))"
```

definition
```
  env_form_map :: "[i,i,i,i]=>i" where
    — map from (environment, formula) pairs to ordinals
   "env_form_map(f,r,A,z)
      == ordermap(list(A) * formula, env_form_r(f,r,A)) ` z"
```

definition
```
  DPow_ord :: "[i,i,i,i,i]=>o" where
    — predicate that holds if k is a valid index for X
   "DPow_ord(f,r,A,X,k) ==
          ∃env ∈ list(A). ∃p ∈ formula.
            arity(p) ≤ succ(length(env)) &
            X = {x∈A. sats(A, p, Cons(x,env))} &
            env_form_map(f,r,A,<env,p>) = k"
```

definition
```
  DPow_least :: "[i,i,i,i]=>i" where
    — function yielding the smallest index for X
   "DPow_least(f,r,A,X) == μ k. DPow_ord(f,r,A,X,k)"
```

definition
```
  DPow_r :: "[i,i,i]=>i" where
    — a wellordering on DPow(A)
   "DPow_r(f,r,A) == measure(DPow(A), DPow_least(f,r,A))"
```

lemma *(in Nat_Times_Nat) well_ord_env_form_r:*
```
    "well_ord(A,r)
     ==> well_ord(list(A) * formula, env_form_r(fn,r,A))"
```

```
by (simp add: env_form_r_def well_ord_rmult well_ord_rlist well_ord_formula)

lemma (in Nat_Times_Nat) Ord_env_form_map:
    "[|well_ord(A,r); z ∈ list(A) * formula|]
     ==> Ord(env_form_map(fn,r,A,z))"
by (simp add: env_form_map_def Ord_ordermap well_ord_env_form_r)

lemma DPow_imp_ex_DPow_ord:
    "X ∈ DPow(A) ==> ∃k. DPow_ord(fn,r,A,X,k)"
apply (simp add: DPow_ord_def)
apply (blast dest!: DPowD)
done

lemma (in Nat_Times_Nat) DPow_ord_imp_Ord:
     "[|DPow_ord(fn,r,A,X,k); well_ord(A,r)|] ==> Ord(k)"
apply (simp add: DPow_ord_def, clarify)
apply (simp add: Ord_env_form_map)
done

lemma (in Nat_Times_Nat) DPow_imp_DPow_least:
    "[|X ∈ DPow(A); well_ord(A,r)|]
     ==> DPow_ord(fn, r, A, X, DPow_least(fn,r,A,X))"
apply (simp add: DPow_least_def)
apply (blast dest: DPow_imp_ex_DPow_ord intro: DPow_ord_imp_Ord LeastI)
done

lemma (in Nat_Times_Nat) env_form_map_inject:
    "[|env_form_map(fn,r,A,u) = env_form_map(fn,r,A,v); well_ord(A,r);
       u ∈ list(A) * formula;  v ∈ list(A) * formula|]
     ==> u=v"
apply (simp add: env_form_map_def)
apply (rule inj_apply_equality [OF bij_is_inj, OF ordermap_bij,
                                OF well_ord_env_form_r], assumption+)
done

lemma (in Nat_Times_Nat) DPow_ord_unique:
    "[|DPow_ord(fn,r,A,X,k); DPow_ord(fn,r,A,Y,k); well_ord(A,r)|]
     ==> X=Y"
apply (simp add: DPow_ord_def, clarify)
apply (drule env_form_map_inject, auto)
done

lemma (in Nat_Times_Nat) well_ord_DPow_r:
    "well_ord(A,r) ==> well_ord(DPow(A), DPow_r(fn,r,A))"
apply (simp add: DPow_r_def)
apply (rule well_ord_measure)
 apply (simp add: DPow_least_def Ord_Least)
apply (drule DPow_imp_DPow_least, assumption)+
apply simp
```

```
apply (blast intro: DPow_ord_unique)
done

lemma (in Nat_Times_Nat) DPow_r_type:
    "DPow_r(fn,r,A) ⊆ DPow(A) * DPow(A)"
by (simp add: DPow_r_def measure_def, blast)
```

15.4 Limit Construction for Well-Orderings

Now we work towards the transfinite definition of wellorderings for *Lset(i)*. We assume as an inductive hypothesis that there is a family of wellorderings for smaller ordinals.

```
definition
  rlimit :: "[i,i=>i]=>i" where
```

— Expresses the wellordering at limit ordinals. The conditional lets us remove the premise *Limit(i)* from some theorems.

```
    "rlimit(i,r) ==
       if Limit(i) then
         {z: Lset(i) * Lset(i).
          ∃x' x. z = <x',x> &
                 (lrank(x') < lrank(x) |
                  (lrank(x') = lrank(x) & <x',x> ∈ r(succ(lrank(x)))))}
       else 0"

definition
  Lset_new :: "i=>i" where
```

— This constant denotes the set of elements introduced at level *succ(i)*

```
    "Lset_new(i) == {x ∈ Lset(succ(i)). lrank(x) = i}"

lemma Limit_Lset_eq2:
    "Limit(i) ==> Lset(i) = (⋃j∈i. Lset_new(j))"
apply (simp add: Limit_Lset_eq)
apply (rule equalityI)
 apply safe
 apply (subgoal_tac "Ord(y)")
  prefer 2 apply (blast intro: Ord_in_Ord Limit_is_Ord)
 apply (simp_all add: Limit_is_Ord Lset_iff_lrank_lt Lset_new_def
                      Ord_mem_iff_lt)
 apply (blast intro: lt_trans)
apply (rule_tac x = "succ(lrank(x))" in bexI)
 apply (simp add: Lset_succ_lrank_iff)
apply (blast intro: Limit_has_succ ltD)
done

lemma wf_on_Lset:
    "wf[Lset(succ(j))](r(succ(j))) ==> wf[Lset_new(j)](rlimit(i,r))"
apply (simp add: wf_on_def Lset_new_def)
apply (erule wf_subset)
apply (simp add: rlimit_def, force)
```

```
done

lemma wf_on_rlimit:
    "(∀j<i. wf[Lset(j)](r(j))) ==> wf[Lset(i)](rlimit(i,r))"
apply (case_tac "Limit(i)")
 prefer 2
 apply (simp add: rlimit_def wf_on_any_0)
apply (simp add: Limit_Lset_eq2)
apply (rule wf_on_Union)
  apply (rule wf_imp_wf_on [OF wf_Memrel [of i]])
 apply (blast intro: wf_on_Lset Limit_has_succ Limit_is_Ord ltI)
apply (force simp add: rlimit_def Limit_is_Ord Lset_iff_lrank_lt Lset_new_def
                       Ord_mem_iff_lt)
done

lemma linear_rlimit:
    "[|Limit(i); ∀j<i. linear(Lset(j), r(j)) |]
     ==> linear(Lset(i), rlimit(i,r))"
apply (frule Limit_is_Ord)
apply (simp add: Limit_Lset_eq2 Lset_new_def)
apply (simp add: linear_def rlimit_def Ball_def lt_Ord Lset_iff_lrank_lt)
apply (simp add: ltI, clarify)
apply (rename_tac u v)
apply (rule_tac i="lrank(u)" and j="lrank(v)" in Ord_linear_lt, simp_all)

apply (drule_tac x="succ(lrank(u) Un lrank(v))" in ospec)
 apply (simp add: ltI)
apply (drule_tac x=u in spec, simp)
apply (drule_tac x=v in spec, simp)
done

lemma well_ord_rlimit:
    "[|Limit(i); ∀j<i. well_ord(Lset(j), r(j)) |]
     ==> well_ord(Lset(i), rlimit(i,r))"
by (blast intro: well_ordI wf_on_rlimit well_ord_is_wf
                         linear_rlimit well_ord_is_linear)

lemma rlimit_cong:
     "(!!j. j<i ==> r'(j) = r(j)) ==> rlimit(i,r) = rlimit(i,r')"
apply (simp add: rlimit_def, clarify)
apply (rule refl iff_refl Collect_cong ex_cong conj_cong)+
apply (simp add: Limit_is_Ord Lset_lrank_lt)
done
```

15.5 Transfinite Definition of the Wellordering on L

```
definition
  L_r :: "[i, i] => i" where
  "L_r(f) == %i.
```

```
transrec3(i, 0, λx r. DPow_r(f, r, Lset(x)),
          λx r. rlimit(x, λy. r`y))"
```

15.5.1 The Corresponding Recursion Equations

```
lemma [simp]: "L_r(f,0) = 0"
by (simp add: L_r_def)

lemma [simp]: "L_r(f, succ(i)) = DPow_r(f, L_r(f,i), Lset(i))"
by (simp add: L_r_def)
```

The limit case is non-trivial because of the distinction between object-level and meta-level abstraction.

```
lemma [simp]: "Limit(i) ==> L_r(f,i) = rlimit(i, L_r(f))"
by (simp cong: rlimit_cong add: transrec3_Limit L_r_def ltD)

lemma (in Nat_Times_Nat) L_r_type:
    "Ord(i) ==> L_r(fn,i) ⊆ Lset(i) * Lset(i)"
apply (induct i rule: trans_induct3_rule)
  apply (simp_all add: Lset_succ DPow_r_type well_ord_DPow_r rlimit_def
                       Transset_subset_DPow [OF Transset_Lset], blast)
done

lemma (in Nat_Times_Nat) well_ord_L_r:
    "Ord(i) ==> well_ord(Lset(i), L_r(fn,i))"
apply (induct i rule: trans_induct3_rule)
apply (simp_all add: well_ord0 Lset_succ L_r_type well_ord_DPow_r
                     well_ord_rlimit ltD)
done

lemma well_ord_L_r:
    "Ord(i) ==> ∃r. well_ord(Lset(i), r)"
apply (insert nat_times_nat_lepoll_nat)
apply (unfold lepoll_def)
apply (blast intro: Nat_Times_Nat.well_ord_L_r Nat_Times_Nat.intro)
done
```

Locale for proving results under the assumption `V=L`

```
locale V_equals_L =
  assumes VL: "L(x)"
```

The Axiom of Choice holds in `L`! Or, to be precise, the Wellordering Theorem.

```
theorem (in V_equals_L) AC: "∃r. well_ord(x,r)"
apply (insert Transset_Lset VL [of x])
apply (simp add: Transset_def L_def)
apply (blast dest!: well_ord_L_r intro: well_ord_subset)
done
```

end

16 Absoluteness for Order Types, Rank Functions and Well-Founded Relations

theory *Rank* **imports** *WF_absolute* **begin**

16.1 Order Types: A Direct Construction by Replacement

```
locale M_ordertype = M_basic +
assumes well_ord_iso_separation:
     "[| M(A); M(f); M(r) |]
      ==> separation (M, λx. x∈A --> (∃y[M]. (∃p[M].
                     fun_apply(M,f,x,y) & pair(M,y,x,p) & p ∈ r)))"
  and obase_separation:
     — part of the order type formalization
     "[| M(A); M(r) |]
      ==> separation(M, λa. ∃x[M]. ∃g[M]. ∃mx[M]. ∃par[M].
             ordinal(M,x) & membership(M,x,mx) & pred_set(M,A,a,r,par)
&
             order_isomorphism(M,par,r,x,mx,g))"
  and obase_equals_separation:
     "[| M(A); M(r) |]
      ==> separation (M, λx. x∈A --> ~(∃y[M]. ∃g[M].
                              ordinal(M,y) & (∃my[M]. ∃pxr[M].
                              membership(M,y,my) & pred_set(M,A,x,r,pxr)
&
                              order_isomorphism(M,pxr,r,y,my,g))))"
  and omap_replacement:
     "[| M(A); M(r) |]
      ==> strong_replacement(M,
             λa z. ∃x[M]. ∃g[M]. ∃mx[M]. ∃par[M].
             ordinal(M,x) & pair(M,a,x,z) & membership(M,x,mx) &
             pred_set(M,A,a,r,par) & order_isomorphism(M,par,r,x,mx,g))"
```

Inductive argument for Kunen's Lemma I 6.1, etc. Simple proof from Halmos, page 72

```
lemma  (in M_ordertype) wellordered_iso_subset_lemma:
     "[| wellordered(M,A,r);  f ∈ ord_iso(A,r, A',r);  A'<= A;  y ∈ A;

       M(A);  M(f);  M(r) |] ==> ~ <f`y, y> ∈ r"
apply (unfold wellordered_def ord_iso_def)
apply (elim conjE CollectE)
apply (erule wellfounded_on_induct, assumption+)
 apply (insert well_ord_iso_separation [of A f r])
 apply (simp, clarify)
apply (drule_tac a = x in bij_is_fun [THEN apply_type], assumption, blast)
```

done

Kunen's Lemma I 6.1, page 14: there's no order-isomorphism to an initial segment of a well-ordering

```
lemma (in M_ordertype) wellordered_iso_predD:
     "[| wellordered(M,A,r);  f ∈ ord_iso(A, r, Order.pred(A,x,r), r);

       M(A);  M(f);  M(r) |] ==> x ∉ A"
apply (rule notI)
apply (frule wellordered_iso_subset_lemma, assumption)
apply (auto elim: predE)

apply (drule ord_iso_is_bij [THEN bij_is_fun, THEN apply_type], assumption)

apply (simp add: Order.pred_def)
done


lemma (in M_ordertype) wellordered_iso_pred_eq_lemma:
     "[| f ∈ ⟨Order.pred(A,y,r), r⟩ ≅ ⟨Order.pred(A,x,r), r⟩;
       wellordered(M,A,r); x∈A; y∈A; M(A); M(f); M(r) |] ==> <x,y> ∉
r"
apply (frule wellordered_is_trans_on, assumption)
apply (rule notI)
apply (drule_tac x2=y and x=x and r2=r in
         wellordered_subset [OF _ pred_subset, THEN wellordered_iso_predD])

apply (simp add: trans_pred_pred_eq)
apply (blast intro: predI dest: transM)+
done
```

Simple consequence of Lemma 6.1

```
lemma (in M_ordertype) wellordered_iso_pred_eq:
     "[| wellordered(M,A,r);
       f ∈ ord_iso(Order.pred(A,a,r), r, Order.pred(A,c,r), r);
       M(A);  M(f);  M(r);  a∈A;  c∈A |] ==> a=c"
apply (frule wellordered_is_trans_on, assumption)
apply (frule wellordered_is_linear, assumption)
apply (erule_tac x=a and y=c in linearE, auto)
apply (drule ord_iso_sym)

apply (blast dest: wellordered_iso_pred_eq_lemma)+
done
```

Following Kunen's Theorem I 7.6, page 17. Note that this material is not required elsewhere.

Can't use *well_ord_iso_preserving* because it needs the strong premise *well_ord(A, r)*

```
lemma (in M_ordertype) ord_iso_pred_imp_lt:
     "[| f ∈ ord_iso(Order.pred(A,x,r), r, i, Memrel(i));
         g ∈ ord_iso(Order.pred(A,y,r), r, j, Memrel(j));
         wellordered(M,A,r);  x ∈ A;  y ∈ A; M(A); M(r); M(f); M(g);
M(j);
         Ord(i); Ord(j); ⟨x,y⟩ ∈ r |]
      ==> i < j"
apply (frule wellordered_is_trans_on, assumption)
apply (frule_tac y=y in transM, assumption)
apply (rule_tac i=i and j=j in Ord_linear_lt, auto)
```

case `i = j` yields a contradiction

```
 apply (rule_tac x1=x and A1="Order.pred(A,y,r)" in
          wellordered_iso_predD [THEN notE])
   apply (blast intro: wellordered_subset [OF _ pred_subset])
  apply (simp add: trans_pred_pred_eq)
  apply (blast intro: Ord_iso_implies_eq ord_iso_sym ord_iso_trans)
 apply (simp_all add: pred_iff pred_closed converse_closed comp_closed)
```

case `j < i` also yields a contradiction

```
apply (frule restrict_ord_iso2, assumption+)
apply (frule ord_iso_sym [THEN ord_iso_is_bij, THEN bij_is_fun])
apply (frule apply_type, blast intro: ltD)
  — thus converse(f) ` j ∈ Order.pred(A, x, r)
apply (simp add: pred_iff)
apply (subgoal_tac
       "∃h[M]. h ∈ ord_iso(Order.pred(A,y,r), r,
                               Order.pred(A, converse(f)`j, r), r)")
 apply (clarify, frule wellordered_iso_pred_eq, assumption+)
 apply (blast dest: wellordered_asym)
apply (intro rexI)
 apply (blast intro: Ord_iso_implies_eq ord_iso_sym ord_iso_trans)+
done


lemma ord_iso_converse1:
     "[| f: ord_iso(A,r,B,s);  <b, f`a>: s;  a:A;  b:B |]
      ==> <converse(f) ` b, a> ∈ r"
apply (frule ord_iso_converse, assumption+)
apply (blast intro: ord_iso_is_bij [THEN bij_is_fun, THEN apply_funtype])

apply (simp add: left_inverse_bij [OF ord_iso_is_bij])
done


definition
  obase :: "[i=>o,i,i] => i" where
       — the domain of om, eventually shown to equal A
   "obase(M,A,r) == {a∈A. ∃x[M]. ∃g[M]. Ord(x) &
```

```
                        g ∈ ord_iso(Order.pred(A,a,r),r,x,Memrel(x))}"

definition
  omap :: "[i=>o,i,i,i] => o" where
    — the function that maps wosets to order types
   "omap(M,A,r,f) ==
       ∀z[M].
        z ∈ f <-> (∃a∈A. ∃x[M]. ∃g[M]. z = <a,x> & Ord(x) &
                        g ∈ ord_iso(Order.pred(A,a,r),r,x,Memrel(x)))"

definition
  otype :: "[i=>o,i,i,i] => o" where — the order types themselves
   "otype(M,A,r,i) == ∃f[M]. omap(M,A,r,f) & is_range(M,f,i)"
```

Can also be proved with the premise `M(z)` instead of `M(f)`, but that version is less useful. This lemma is also more useful than the definition, `omap_def`.

```
lemma (in M_ordertype) omap_iff:
     "[| omap(M,A,r,f); M(A); M(f) |]
      ==> z ∈ f <->
          (∃a∈A. ∃x[M]. ∃g[M]. z = <a,x> & Ord(x) &
                                g ∈ ord_iso(Order.pred(A,a,r),r,x,Memrel(x)))"
apply (simp add: omap_def Memrel_closed pred_closed)
apply (rule iffI)
 apply (drule_tac [2] x=z in rspec)
 apply (drule_tac x=z in rspec)
 apply (blast dest: transM)+
done

lemma (in M_ordertype) omap_unique:
     "[| omap(M,A,r,f); omap(M,A,r,f'); M(A); M(r); M(f); M(f') |] ==>
f' = f"
apply (rule equality_iffI)
apply (simp add: omap_iff)
done

lemma (in M_ordertype) omap_yields_Ord:
     "[| omap(M,A,r,f); ⟨a,x⟩ ∈ f; M(a); M(x) |]  ==> Ord(x)"
  by (simp add: omap_def)

lemma (in M_ordertype) otype_iff:
     "[| otype(M,A,r,i); M(A); M(r); M(i) |]
      ==> x ∈ i <->
          (M(x) & Ord(x) &
           (∃a∈A. ∃g[M]. g ∈ ord_iso(Order.pred(A,a,r),r,x,Memrel(x))))"
apply (auto simp add: omap_iff otype_def)
 apply (blast intro: transM)
apply (rule rangeI)
apply (frule transM, assumption)
apply (simp add: omap_iff, blast)
```

```
done

lemma (in M_ordertype) otype_eq_range:
     "[| omap(M,A,r,f); otype(M,A,r,i); M(A); M(r); M(f); M(i) |]
      ==> i = range(f)"
apply (auto simp add: otype_def omap_iff)
apply (blast dest: omap_unique)
done


lemma (in M_ordertype) Ord_otype:
     "[| otype(M,A,r,i); trans[A](r); M(A); M(r); M(i) |] ==> Ord(i)"
apply (rule OrdI)
prefer 2
    apply (simp add: Ord_def otype_def omap_def)
    apply clarify
    apply (frule pair_components_in_M, assumption)
    apply blast
apply (auto simp add: Transset_def otype_iff)
  apply (blast intro: transM)
 apply (blast intro: Ord_in_Ord)
apply (rename_tac y a g)
apply (frule ord_iso_sym [THEN ord_iso_is_bij, THEN bij_is_fun,
                          THEN apply_funtype],  assumption)
apply (rule_tac x="converse(g)`y" in bexI)
 apply (frule_tac a="converse(g) ` y" in ord_iso_restrict_pred, assumption)

apply (safe elim!: predE)
apply (blast intro: restrict_ord_iso ord_iso_sym ltI dest: transM)
done

lemma (in M_ordertype) domain_omap:
     "[| omap(M,A,r,f);  M(A); M(r); M(B); M(f) |]
      ==> domain(f) = obase(M,A,r)"
apply (simp add: domain_closed obase_def)
apply (rule equality_iffI)
apply (simp add: domain_iff omap_iff, blast)
done

lemma (in M_ordertype) omap_subset:
     "[| omap(M,A,r,f); otype(M,A,r,i);
       M(A); M(r); M(f); M(B); M(i) |] ==> f ⊆ obase(M,A,r) * i"
apply clarify
apply (simp add: omap_iff obase_def)
apply (force simp add: otype_iff)
done

lemma (in M_ordertype) omap_funtype:
     "[| omap(M,A,r,f); otype(M,A,r,i);
```

```
        M(A); M(r); M(f); M(i) |] ==> f ∈ obase(M,A,r) -> i"
apply (simp add: domain_omap omap_subset Pi_iff function_def omap_iff)

apply (blast intro: Ord_iso_implies_eq ord_iso_sym ord_iso_trans)
done
```

```
lemma (in M_ordertype) wellordered_omap_bij:
     "[| wellordered(M,A,r); omap(M,A,r,f); otype(M,A,r,i);
       M(A); M(r); M(f); M(i) |] ==> f ∈ bij(obase(M,A,r),i)"
apply (insert omap_funtype [of A r f i])
apply (auto simp add: bij_def inj_def)
prefer 2  apply (blast intro: fun_is_surj dest: otype_eq_range)
apply (frule_tac a=w in apply_Pair, assumption)
apply (frule_tac a=x in apply_Pair, assumption)
apply (simp add: omap_iff)
apply (blast intro: wellordered_iso_pred_eq ord_iso_sym ord_iso_trans)

done
```

This is not the final result: we must show `oB(A, r) = A`

```
lemma (in M_ordertype) omap_ord_iso:
     "[| wellordered(M,A,r); omap(M,A,r,f); otype(M,A,r,i);
       M(A); M(r); M(f); M(i) |] ==> f ∈ ord_iso(obase(M,A,r),r,i,Memrel(i))"
apply (rule ord_isoI)
 apply (erule wellordered_omap_bij, assumption+)
apply (insert omap_funtype [of A r f i], simp)
apply (frule_tac a=x in apply_Pair, assumption)
apply (frule_tac a=y in apply_Pair, assumption)
apply (auto simp add: omap_iff)
```

direction 1: assuming `⟨x, y⟩ ∈ r`

```
 apply (blast intro: ltD ord_iso_pred_imp_lt)
```

direction 2: proving `⟨x, y⟩ ∈ r` using linearity of `r`

```
apply (rename_tac x y g ga)
apply (frule wellordered_is_linear, assumption,
       erule_tac x=x and y=y in linearE, assumption+)
```

the case `x = y` leads to immediate contradiction

```
apply (blast elim: mem_irrefl)
```

the case `⟨y, x⟩ ∈ r`: handle like the opposite direction

```
apply (blast dest: ord_iso_pred_imp_lt ltD elim: mem_asym)
done
```

```
lemma (in M_ordertype) Ord_omap_image_pred:
     "[| wellordered(M,A,r); omap(M,A,r,f); otype(M,A,r,i);
       M(A); M(r); M(f); M(i); b ∈ A |] ==> Ord(f `` Order.pred(A,b,r))"
```

```
apply (frule wellordered_is_trans_on, assumption)
apply (rule OrdI)
        prefer 2 apply (simp add: image_iff omap_iff Ord_def, blast)
```

Hard part is to show that the image is a transitive set.

```
apply (simp add: Transset_def, clarify)
apply (simp add: image_iff pred_iff apply_iff [OF omap_funtype [of A r
f i]])
apply (rename_tac c j, clarify)
apply (frule omap_funtype [of A r f, THEN apply_funtype], assumption+)
apply (subgoal_tac "j ∈ i")
        prefer 2 apply (blast intro: Ord_trans Ord_otype)
apply (subgoal_tac "converse(f) ` j ∈ obase(M,A,r)")
        prefer 2
        apply (blast dest: wellordered_omap_bij [THEN bij_converse_bij,

                                          THEN bij_is_fun, THEN apply_funtype])
apply (rule_tac x="converse(f) ` j" in bexI)
 apply (simp add: right_inverse_bij [OF wellordered_omap_bij])
apply (intro predI conjI)
 apply (erule_tac b=c in trans_onD)
 apply (rule ord_iso_converse1 [OF omap_ord_iso [of A r f i]])
apply (auto simp add: obase_def)
done

lemma (in M_ordertype) restrict_omap_ord_iso:
     "[| wellordered(M,A,r); omap(M,A,r,f); otype(M,A,r,i);
       D ⊆ obase(M,A,r); M(A); M(r); M(f); M(i) |]
      ==> restrict(f,D) ∈ (⟨D,r⟩ ≅ ⟨f``D, Memrel(f``D)⟩)"
apply (frule ord_iso_restrict_image [OF omap_ord_iso [of A r f i]],
       assumption+)
apply (drule ord_iso_sym [THEN subset_ord_iso_Memrel])
apply (blast dest: subsetD [OF omap_subset])
apply (drule ord_iso_sym, simp)
done

lemma (in M_ordertype) obase_equals:
     "[| wellordered(M,A,r); omap(M,A,r,f); otype(M,A,r,i);
       M(A); M(r); M(f); M(i) |] ==> obase(M,A,r) = A"
apply (rule equalityI, force simp add: obase_def, clarify)
apply (unfold obase_def, simp)
apply (frule wellordered_is_wellfounded_on, assumption)
apply (erule wellfounded_on_induct, assumption+)
 apply (frule obase_equals_separation [of A r], assumption)
 apply (simp, clarify)
apply (rename_tac b)
apply (subgoal_tac "Order.pred(A,b,r) <= obase(M,A,r)")
 apply (blast intro!: restrict_omap_ord_iso Ord_omap_image_pred)
apply (force simp add: pred_iff obase_def)
```

```
done
```

Main result: *om* gives the order-isomorphism $\langle$A, r$\rangle \cong \langle$i, Memrel(i)$\rangle$

```
theorem (in M_ordertype) omap_ord_iso_otype:
     "[| wellordered(M,A,r); omap(M,A,r,f); otype(M,A,r,i);
       M(A); M(r); M(f); M(i) |] ==> f ∈ ord_iso(A, r, i, Memrel(i))"
apply (frule omap_ord_iso, assumption+)
apply (simp add: obase_equals)
done

lemma (in M_ordertype) obase_exists:
     "[| M(A); M(r) |] ==> M(obase(M,A,r))"
apply (simp add: obase_def)
apply (insert obase_separation [of A r])
apply (simp add: separation_def)
done

lemma (in M_ordertype) omap_exists:
     "[| M(A); M(r) |] ==> ∃z[M]. omap(M,A,r,z)"
apply (simp add: omap_def)
apply (insert omap_replacement [of A r])
apply (simp add: strong_replacement_def)
apply (drule_tac x="obase(M,A,r)" in rspec)
 apply (simp add: obase_exists)
apply (simp add: Memrel_closed pred_closed obase_def)
apply (erule impE)
 apply (clarsimp simp add: univalent_def)
 apply (blast intro: Ord_iso_implies_eq ord_iso_sym ord_iso_trans, clarify)

apply (rule_tac x=Y in rexI)
apply (simp add: Memrel_closed pred_closed obase_def, blast, assumption)
done

declare rall_simps [simp] rex_simps [simp]

lemma (in M_ordertype) otype_exists:
     "[| wellordered(M,A,r); M(A); M(r) |] ==> ∃i[M]. otype(M,A,r,i)"
apply (insert omap_exists [of A r])
apply (simp add: otype_def, safe)
apply (rule_tac x="range(x)" in rexI)
apply blast+
done

lemma (in M_ordertype) ordertype_exists:
     "[| wellordered(M,A,r); M(A); M(r) |]
      ==> ∃f[M]. (∃i[M]. Ord(i) & f ∈ ord_iso(A, r, i, Memrel(i)))"
apply (insert obase_exists [of A r] omap_exists [of A r] otype_exists
[of A r], simp, clarify)
apply (rename_tac i)
```

```
apply (subgoal_tac "Ord(i)", blast intro: omap_ord_iso_otype)
apply (rule Ord_otype)
    apply (force simp add: otype_def range_closed)
   apply (simp_all add: wellordered_is_trans_on)
done
```

```
lemma (in M_ordertype) relativized_imp_well_ord:
     "[| wellordered(M,A,r); M(A); M(r) |] ==> well_ord(A,r)"
apply (insert ordertype_exists [of A r], simp)
apply (blast intro: well_ord_ord_iso well_ord_Memrel)
done
```

16.2 Kunen's theorem 5.4, page 127

(a) The notion of Wellordering is absolute

```
theorem (in M_ordertype) well_ord_abs [simp]:
     "[| M(A); M(r) |] ==> wellordered(M,A,r) <-> well_ord(A,r)"
by (blast intro: well_ord_imp_relativized relativized_imp_well_ord)
```

(b) Order types are absolute

```
theorem (in M_ordertype)
     "[| wellordered(M,A,r); f ∈ ord_iso(A, r, i, Memrel(i));
       M(A); M(r); M(f); M(i); Ord(i) |] ==> i = ordertype(A,r)"
by (blast intro: Ord_ordertype relativized_imp_well_ord ordertype_ord_iso
                 Ord_iso_implies_eq ord_iso_sym ord_iso_trans)
```

16.3 Ordinal Arithmetic: Two Examples of Recursion

Note: the remainder of this theory is not needed elsewhere.

16.3.1 Ordinal Addition

```
definition
  is_oadd_fun :: "[i=>o,i,i,i,i] => o" where
    "is_oadd_fun(M,i,j,x,f) ==
       (∀ sj msj. M(sj) --> M(msj) -->
                 successor(M,j,sj) --> membership(M,sj,msj) -->
                 M_is_recfun(M,
                     %x g y. ∃gx[M]. image(M,g,x,gx) & union(M,i,gx,y),
                     msj, x, f))"
```

```
definition
  is_oadd :: "[i=>o,i,i,i] => o" where
    "is_oadd(M,i,j,k) ==
        (~ ordinal(M,i) & ~ ordinal(M,j) & k=0) |
        (~ ordinal(M,i) & ordinal(M,j) & k=j) |
        (ordinal(M,i) & ~ ordinal(M,j) & k=i) |
```

```
        (ordinal(M,i) & ordinal(M,j) &
         (∃f fj sj. M(f) & M(fj) & M(sj) &
                    successor(M,j,sj) & is_oadd_fun(M,i,sj,sj,f) &
                    fun_apply(M,f,j,fj) & fj = k))"
```

definition

```
  omult_eqns :: "[i,i,i,i] => o" where
    "omult_eqns(i,x,g,z) ==
            Ord(x) &
            (x=0 --> z=0) &
            (∀j. x = succ(j) --> z = g`j ++ i) &
            (Limit(x) --> z = ⋃(g``x))"
```

definition

```
  is_omult_fun :: "[i=>o,i,i,i] => o" where
    "is_omult_fun(M,i,j,f) ==
            (∃df. M(df) & is_function(M,f) &
                  is_domain(M,f,df) & subset(M, j, df)) &
            (∀x∈j. omult_eqns(i,x,f,f`x))"
```

definition

```
  is_omult :: "[i=>o,i,i,i] => o" where
    "is_omult(M,i,j,k) ==
        ∃f fj sj. M(f) & M(fj) & M(sj) &
                  successor(M,j,sj) & is_omult_fun(M,i,sj,f) &
                  fun_apply(M,f,j,fj) & fj = k"
```

```
locale M_ord_arith = M_ordertype +
  assumes oadd_strong_replacement:
   "[| M(i); M(j) |] ==>
    strong_replacement(M,
         λx z. ∃y[M]. pair(M,x,y,z) &
                  (∃f[M]. ∃fx[M]. is_oadd_fun(M,i,j,x,f) &
                           image(M,f,x,fx) & y = i Un fx))"

 and omult_strong_replacement':
   "[| M(i); M(j) |] ==>
    strong_replacement(M,
         λx z. ∃y[M]. z = <x,y> &
             (∃g[M]. is_recfun(Memrel(succ(j)),x,%x g. THE z. omult_eqns(i,x,g,z),g)
&
             y = (THE z. omult_eqns(i, x, g, z))))"
```

is_oadd_fun: Relating the pure "language of set theory" to Isabelle/ZF

```
lemma (in M_ord_arith) is_oadd_fun_iff:
   "[| a≤j; M(i); M(j); M(a); M(f) |]
    ==> is_oadd_fun(M,i,j,a,f) <->
```

```
        f ∈ a → range(f) & (∀x. M(x) --> x < a --> f`x = i Un f``x)"
apply (frule lt_Ord)
apply (simp add: is_oadd_fun_def Memrel_closed Un_closed
              relation2_def is_recfun_abs [of "%x g. i Un g``x"]
              image_closed is_recfun_iff_equation
              Ball_def lt_trans [OF ltI, of _ a] lt_Memrel)
apply (simp add: lt_def)
apply (blast dest: transM)
done


lemma (in M_ord_arith) oadd_strong_replacement':
    "[| M(i); M(j) |] ==>
     strong_replacement(M,
            λx z. ∃y[M]. z = <x,y> &
                  (∃g[M]. is_recfun(Memrel(succ(j)),x,%x g. i Un g``x,g)
&
                  y = i Un g``x))"
apply (insert oadd_strong_replacement [of i j])
apply (simp add: is_oadd_fun_def relation2_def
                 is_recfun_abs [of "%x g. i Un g``x"])
done


lemma (in M_ord_arith) exists_oadd:
    "[| Ord(j);  M(i);  M(j) |]
     ==> ∃f[M]. is_recfun(Memrel(succ(j)), j, %x g. i Un g``x, f)"
apply (rule wf_exists_is_recfun [OF wf_Memrel trans_Memrel])
    apply (simp_all add: Memrel_type oadd_strong_replacement')
done

lemma (in M_ord_arith) exists_oadd_fun:
    "[| Ord(j);  M(i);  M(j) |] ==> ∃f[M]. is_oadd_fun(M,i,succ(j),succ(j),f)"
apply (rule exists_oadd [THEN rexE])
apply (erule Ord_succ, assumption, simp)
apply (rename_tac f)
apply (frule is_recfun_type)
apply (rule_tac x=f in rexI)
 apply (simp add: fun_is_function domain_of_fun lt_Memrel apply_recfun
lt_def
                  is_oadd_fun_iff Ord_trans [OF _ succI1], assumption)
done

lemma (in M_ord_arith) is_oadd_fun_apply:
    "[| x < j; M(i); M(j); M(f); is_oadd_fun(M,i,j,j,f) |]
     ==> f`x = i Un (⋃k∈x. {f ` k})"
apply (simp add: is_oadd_fun_iff lt_Ord2, clarify)
apply (frule lt_closed, simp)
apply (frule leI [THEN le_imp_subset])
```

```
apply (simp add: image_fun, blast)
done

lemma (in M_ord_arith) is_oadd_fun_iff_oadd [rule_format]:
    "[| is_oadd_fun(M,i,J,J,f); M(i); M(J); M(f); Ord(i); Ord(j) |]
     ==> j<J --> f`j = i++j"
apply (erule_tac i=j in trans_induct, clarify)
apply (subgoal_tac "∀k∈x. k<J")
 apply (simp (no_asm_simp) add: is_oadd_def oadd_unfold is_oadd_fun_apply)
apply (blast intro: lt_trans ltI lt_Ord)
done

lemma (in M_ord_arith) Ord_oadd_abs:
    "[| M(i); M(j); M(k); Ord(i); Ord(j) |] ==> is_oadd(M,i,j,k) <-> k
= i++j"
apply (simp add: is_oadd_def is_oadd_fun_iff_oadd)
apply (frule exists_oadd_fun [of j i], blast+)
done

lemma (in M_ord_arith) oadd_abs:
    "[| M(i); M(j); M(k) |] ==> is_oadd(M,i,j,k) <-> k = i++j"
apply (case_tac "Ord(i) & Ord(j)")
 apply (simp add: Ord_oadd_abs)
apply (auto simp add: is_oadd_def oadd_eq_if_raw_oadd)
done

lemma (in M_ord_arith) oadd_closed [intro,simp]:
    "[| M(i); M(j) |] ==> M(i++j)"
apply (simp add: oadd_eq_if_raw_oadd, clarify)
apply (simp add: raw_oadd_eq_oadd)
apply (frule exists_oadd_fun [of j i], auto)
apply (simp add: apply_closed is_oadd_fun_iff_oadd [symmetric])
done
```

16.3.2 Ordinal Multiplication

```
lemma omult_eqns_unique:
     "[| omult_eqns(i,x,g,z); omult_eqns(i,x,g,z') |] ==> z=z'"
apply (simp add: omult_eqns_def, clarify)
apply (erule Ord_cases, simp_all)
done

lemma omult_eqns_0: "omult_eqns(i,0,g,z) <-> z=0"
by (simp add: omult_eqns_def)

lemma the_omult_eqns_0: "(THE z. omult_eqns(i,0,g,z)) = 0"
by (simp add: omult_eqns_0)

lemma omult_eqns_succ: "omult_eqns(i,succ(j),g,z) <-> Ord(j) & z = g`j
```

```
++ i"
by (simp add: omult_eqns_def)

lemma the_omult_eqns_succ:
     "Ord(j) ==> (THE z. omult_eqns(i,succ(j),g,z)) = g`j ++ i"
by (simp add: omult_eqns_succ)

lemma omult_eqns_Limit:
     "Limit(x) ==> omult_eqns(i,x,g,z) <-> z = ⋃(g``x)"
apply (simp add: omult_eqns_def)
apply (blast intro: Limit_is_Ord)
done

lemma the_omult_eqns_Limit:
     "Limit(x) ==> (THE z. omult_eqns(i,x,g,z)) = ⋃(g``x)"
by (simp add: omult_eqns_Limit)

lemma omult_eqns_Not: "~ Ord(x) ==> ~ omult_eqns(i,x,g,z)"
by (simp add: omult_eqns_def)


lemma (in M_ord_arith) the_omult_eqns_closed:
    "[| M(i); M(x); M(g); function(g) |]
     ==> M(THE z. omult_eqns(i, x, g, z))"
apply (case_tac "Ord(x)")
 prefer 2 apply (simp add: omult_eqns_Not) — trivial, non-Ord case
apply (erule Ord_cases)
  apply (simp add: omult_eqns_0)
 apply (simp add: omult_eqns_succ apply_closed oadd_closed)
apply (simp add: omult_eqns_Limit)
done

lemma (in M_ord_arith) exists_omult:
    "[| Ord(j);  M(i);  M(j) |]
     ==> ∃f[M]. is_recfun(Memrel(succ(j)), j, %x g. THE z. omult_eqns(i,x,g,z),
f)"
apply (rule wf_exists_is_recfun [OF wf_Memrel trans_Memrel])
    apply (simp_all add: Memrel_type omult_strong_replacement')
apply (blast intro: the_omult_eqns_closed)
done

lemma (in M_ord_arith) exists_omult_fun:
    "[| Ord(j);  M(i);  M(j) |] ==> ∃f[M]. is_omult_fun(M,i,succ(j),f)"
apply (rule exists_omult [THEN rexE])
apply (erule Ord_succ, assumption, simp)
apply (rename_tac f)
apply (frule is_recfun_type)
apply (rule_tac x=f in rexI)
apply (simp add: fun_is_function domain_of_fun lt_Memrel apply_recfun
```

```
lt_def
                 is_omult_fun_def Ord_trans [OF _ succI1])
 apply (force dest: Ord_in_Ord'
              simp add: omult_eqns_def the_omult_eqns_0 the_omult_eqns_succ
                        the_omult_eqns_Limit, assumption)
done

lemma (in M_ord_arith) is_omult_fun_apply_0:
    "[| 0 < j; is_omult_fun(M,i,j,f) |] ==> f`0 = 0"
by (simp add: is_omult_fun_def omult_eqns_def lt_def ball_conj_distrib)

lemma (in M_ord_arith) is_omult_fun_apply_succ:
    "[| succ(x) < j; is_omult_fun(M,i,j,f) |] ==> f`succ(x) = f`x ++ i"
by (simp add: is_omult_fun_def omult_eqns_def lt_def, blast)

lemma (in M_ord_arith) is_omult_fun_apply_Limit:
    "[| x < j; Limit(x); M(j); M(f); is_omult_fun(M,i,j,f) |]
     ==> f ` x = (⋃y∈x. f`y)"
apply (simp add: is_omult_fun_def omult_eqns_def domain_closed lt_def,
clarify)
apply (drule subset_trans [OF OrdmemD], assumption+)
apply (simp add: ball_conj_distrib omult_Limit image_function)
done

lemma (in M_ord_arith) is_omult_fun_eq_omult:
    "[| is_omult_fun(M,i,J,f); M(J); M(f); Ord(i); Ord(j) |]
     ==> j<J --> f`j = i**j"
apply (erule_tac i=j in trans_induct3)
apply (safe del: impCE)
  apply (simp add: is_omult_fun_apply_0)
 apply (subgoal_tac "x<J")
  apply (simp add: is_omult_fun_apply_succ omult_succ)
 apply (blast intro: lt_trans)
apply (subgoal_tac "∀k∈x. k<J")
 apply (simp add: is_omult_fun_apply_Limit omult_Limit)
apply (blast intro: lt_trans ltI lt_Ord)
done

lemma (in M_ord_arith) omult_abs:
    "[| M(i); M(j); M(k); Ord(i); Ord(j) |] ==> is_omult(M,i,j,k) <->
k = i**j"
apply (simp add: is_omult_def is_omult_fun_eq_omult)
apply (frule exists_omult_fun [of j i], blast+)
done
```

16.4 Absoluteness of Well-Founded Relations

Relativized to `M`: Every well-founded relation is a subset of some inverse image of an ordinal. Key step is the construction (in `M`) of a rank function.

```
locale M_wfrank = M_trancl +
  assumes wfrank_separation:
     "M(r) ==>
      separation (M, λx.
         ∀rplus[M]. tran_closure(M,r,rplus) -->
         ~ (∃f[M]. M_is_recfun(M, %x f y. is_range(M,f,y), rplus, x,
f)))"
 and wfrank_strong_replacement:
     "M(r) ==>
      strong_replacement(M, λx z.
         ∀rplus[M]. tran_closure(M,r,rplus) -->
         (∃y[M]. ∃f[M]. pair(M,x,y,z)  &
                       M_is_recfun(M, %x f y. is_range(M,f,y), rplus,
x, f) &
                       is_range(M,f,y)))"
 and Ord_wfrank_separation:
     "M(r) ==>
      separation (M, λx.
         ∀rplus[M]. tran_closure(M,r,rplus) -->
          ~ (∀f[M]. ∀rangef[M].
             is_range(M,f,rangef) -->
             M_is_recfun(M, λx f y. is_range(M,f,y), rplus, x, f) -->
             ordinal(M,rangef)))"
```

Proving that the relativized instances of Separation or Replacement agree with the "real" ones.

```
lemma (in M_wfrank) wfrank_separation':
     "M(r) ==>
      separation
           (M, λx. ~ (∃f[M]. is_recfun(r^+, x, %x f. range(f), f)))"
apply (insert wfrank_separation [of r])
apply (simp add: relation2_def is_recfun_abs [of "%x. range"])
done

lemma (in M_wfrank) wfrank_strong_replacement':
     "M(r) ==>
      strong_replacement(M, λx z. ∃y[M]. ∃f[M].
                  pair(M,x,y,z) & is_recfun(r^+, x, %x f. range(f), f)
&
                  y = range(f))"
apply (insert wfrank_strong_replacement [of r])
apply (simp add: relation2_def is_recfun_abs [of "%x. range"])
done

lemma (in M_wfrank) Ord_wfrank_separation':
     "M(r) ==>
      separation (M, λx.
         ~ (∀f[M]. is_recfun(r^+, x, λx. range, f) --> Ord(range(f))))"
```

```
apply (insert Ord_wfrank_separation [of r])
apply (simp add: relation2_def is_recfun_abs [of "%x. range"])
done
```

This function, defined using replacement, is a rank function for well-founded relations within the class M.

```
definition
  wellfoundedrank :: "[i=>o,i,i] => i" where
    "wellfoundedrank(M,r,A) ==
        {p. x∈A, ∃y[M]. ∃f[M].
                       p = <x,y> & is_recfun(r^+, x, %x f. range(f), f)
&
                       y = range(f)}"

lemma (in M_wfrank) exists_wfrank:
    "[| wellfounded(M,r); M(a); M(r) |]
     ==> ∃f[M]. is_recfun(r^+, a, %x f. range(f), f)"
apply (rule wellfounded_exists_is_recfun)
      apply (blast intro: wellfounded_trancl)
     apply (rule trans_trancl)
    apply (erule wfrank_separation')
   apply (erule wfrank_strong_replacement')
apply (simp_all add: trancl_subset_times)
done

lemma (in M_wfrank) M_wellfoundedrank:
    "[| wellfounded(M,r); M(r); M(A) |] ==> M(wellfoundedrank(M,r,A))"
apply (insert wfrank_strong_replacement' [of r])
apply (simp add: wellfoundedrank_def)
apply (rule strong_replacement_closed)
   apply assumption+
 apply (rule univalent_is_recfun)
   apply (blast intro: wellfounded_trancl)
  apply (rule trans_trancl)
 apply (simp add: trancl_subset_times)
apply (blast dest: transM)
done

lemma (in M_wfrank) Ord_wfrank_range [rule_format]:
    "[| wellfounded(M,r); a∈A; M(r); M(A) |]
     ==> ∀f[M]. is_recfun(r^+, a, %x f. range(f), f) --> Ord(range(f))"
apply (drule wellfounded_trancl, assumption)
apply (rule wellfounded_induct, assumption, erule (1) transM)
  apply simp
 apply (blast intro: Ord_wfrank_separation', clarify)
```

The reasoning in both cases is that we get y such that $\langle y,\ x\rangle \in$ `r^+`. We find that `f ‘ y = restrict(f, r^+ -‘‘ {y})`.

```
apply (rule OrdI [OF _ Ord_is_Transset])
```

An ordinal is a transitive set...

```
 apply (simp add: Transset_def)
 apply clarify
 apply (frule apply_recfun2, assumption)
 apply (force simp add: restrict_iff)
```

...of ordinals. This second case requires the induction hyp.

```
apply clarify
apply (rename_tac i y)
apply (frule apply_recfun2, assumption)
apply (frule is_recfun_imp_in_r, assumption)
apply (frule is_recfun_restrict)

    apply (simp add: trans_trancl trancl_subset_times)+
apply (drule spec [THEN mp], assumption)
apply (subgoal_tac "M(restrict(f, r^+ -`` {y}))")
 apply (drule_tac x="restrict(f, r^+ -`` {y})" in rspec)
apply assumption
 apply (simp add: function_apply_equality [OF _ is_recfun_imp_function])
apply (blast dest: pair_components_in_M)
done

lemma (in M_wfrank) Ord_range_wellfoundedrank:
    "[| wellfounded(M,r); r ⊆ A*A;  M(r); M(A) |]
     ==> Ord (range(wellfoundedrank(M,r,A)))"
apply (frule wellfounded_trancl, assumption)
apply (frule trancl_subset_times)
apply (simp add: wellfoundedrank_def)
apply (rule OrdI [OF _ Ord_is_Transset])
 prefer 2
```

by our previous result the range consists of ordinals.

```
 apply (blast intro: Ord_wfrank_range)
```

We still must show that the range is a transitive set.

```
apply (simp add: Transset_def, clarify, simp)
apply (rename_tac x i f u)
apply (frule is_recfun_imp_in_r, assumption)
apply (subgoal_tac "M(u) & M(i) & M(x)")
 prefer 2 apply (blast dest: transM, clarify)
apply (rule_tac a=u in rangeI)
apply (rule_tac x=u in ReplaceI)
  apply simp
  apply (rule_tac x="restrict(f, r^+ -`` {u})" in rexI)
   apply (blast intro: is_recfun_restrict trans_trancl dest: apply_recfun2)
  apply simp
apply blast
```

Unicity requirement of Replacement

```
apply clarify
apply (frule apply_recfun2, assumption)
apply (simp add: trans_trancl is_recfun_cut)
done

lemma (in M_wfrank) function_wellfoundedrank:
    "[| wellfounded(M,r); M(r); M(A)|]
     ==> function(wellfoundedrank(M,r,A))"
apply (simp add: wellfoundedrank_def function_def, clarify)
```

Uniqueness: repeated below!

```
apply (drule is_recfun_functional, assumption)
     apply (blast intro: wellfounded_trancl)
    apply (simp_all add: trancl_subset_times trans_trancl)
done

lemma (in M_wfrank) domain_wellfoundedrank:
    "[| wellfounded(M,r); M(r); M(A)|]
     ==> domain(wellfoundedrank(M,r,A)) = A"
apply (simp add: wellfoundedrank_def function_def)
apply (rule equalityI, auto)
apply (frule transM, assumption)
apply (frule_tac a=x in exists_wfrank, assumption+, clarify)
apply (rule_tac b="range(f)" in domainI)
apply (rule_tac x=x in ReplaceI)
  apply simp
  apply (rule_tac x=f in rexI, blast, simp_all)
```

Uniqueness (for Replacement): repeated above!

```
apply clarify
apply (drule is_recfun_functional, assumption)
    apply (blast intro: wellfounded_trancl)
    apply (simp_all add: trancl_subset_times trans_trancl)
done

lemma (in M_wfrank) wellfoundedrank_type:
    "[| wellfounded(M,r);  M(r); M(A)|]
     ==> wellfoundedrank(M,r,A) ∈ A -> range(wellfoundedrank(M,r,A))"
apply (frule function_wellfoundedrank [of r A], assumption+)
apply (frule function_imp_Pi)
 apply (simp add: wellfoundedrank_def relation_def)
 apply blast
apply (simp add: domain_wellfoundedrank)
done

lemma (in M_wfrank) Ord_wellfoundedrank:
    "[| wellfounded(M,r); a ∈ A; r ⊆ A*A;  M(r); M(A) |]
     ==> Ord(wellfoundedrank(M,r,A) ` a)"
by (blast intro: apply_funtype [OF wellfoundedrank_type]
```

```
                  Ord_in_Ord [OF Ord_range_wellfoundedrank])

lemma (in M_wfrank) wellfoundedrank_eq:
     "[| is_recfun(r^+, a, %x. range, f);
         wellfounded(M,r);  a ∈ A; M(f); M(r); M(A)|]
      ==> wellfoundedrank(M,r,A) ` a = range(f)"
apply (rule apply_equality)
 prefer 2 apply (blast intro: wellfoundedrank_type)
apply (simp add: wellfoundedrank_def)
apply (rule ReplaceI)
  apply (rule_tac x="range(f)" in rexI)
  apply blast
 apply simp_all
```

Unicity requirement of Replacement

```
apply clarify
apply (drule is_recfun_functional, assumption)
    apply (blast intro: wellfounded_trancl)
    apply (simp_all add: trancl_subset_times trans_trancl)
done


lemma (in M_wfrank) wellfoundedrank_lt:
     "[| <a,b> ∈ r;
         wellfounded(M,r); r ⊆ A*A;  M(r); M(A)|]
      ==> wellfoundedrank(M,r,A) ` a < wellfoundedrank(M,r,A) ` b"
apply (frule wellfounded_trancl, assumption)
apply (subgoal_tac "a∈A & b∈A")
 prefer 2 apply blast
apply (simp add: lt_def Ord_wellfoundedrank, clarify)
apply (frule exists_wfrank [of concl: _ b], erule (1) transM, assumption)
apply clarify
apply (rename_tac fb)
apply (frule is_recfun_restrict [of concl: "r^+" a])
    apply (rule trans_trancl, assumption)
   apply (simp_all add: r_into_trancl trancl_subset_times)
```

Still the same goal, but with new `is_recfun` assumptions.

```
apply (simp add: wellfoundedrank_eq)
apply (frule_tac a=a in wellfoundedrank_eq, assumption+)
   apply (simp_all add: transM [of a])
```

We have used equations for wellfoundedrank and now must use some for `is_recfun`.

```
apply (rule_tac a=a in rangeI)
apply (simp add: is_recfun_type [THEN apply_iff] vimage_singleton_iff
                 r_into_trancl apply_recfun r_into_trancl)
done
```

```
lemma (in M_wfrank) wellfounded_imp_subset_rvimage:
     "[|wellfounded(M,r); r ⊆ A*A; M(r); M(A)|]
      ==> ∃i f. Ord(i) & r <= rvimage(A, f, Memrel(i))"
apply (rule_tac x="range(wellfoundedrank(M,r,A))" in exI)
apply (rule_tac x="wellfoundedrank(M,r,A)" in exI)
apply (simp add: Ord_range_wellfoundedrank, clarify)
apply (frule subsetD, assumption, clarify)
apply (simp add: rvimage_iff wellfoundedrank_lt [THEN ltD])
apply (blast intro: apply_rangeI wellfoundedrank_type)
done

lemma (in M_wfrank) wellfounded_imp_wf:
     "[|wellfounded(M,r); relation(r); M(r)|] ==> wf(r)"
by (blast dest!: relation_field_times_field wellfounded_imp_subset_rvimage
          intro: wf_rvimage_Ord [THEN wf_subset])

lemma (in M_wfrank) wellfounded_on_imp_wf_on:
     "[|wellfounded_on(M,A,r); relation(r); M(r); M(A)|] ==> wf[A](r)"
apply (simp add: wellfounded_on_iff_wellfounded wf_on_def)
apply (rule wellfounded_imp_wf)
apply (simp_all add: relation_def)
done


theorem (in M_wfrank) wf_abs:
     "[|relation(r); M(r)|] ==> wellfounded(M,r) <-> wf(r)"
by (blast intro: wellfounded_imp_wf wf_imp_relativized)

theorem (in M_wfrank) wf_on_abs:
     "[|relation(r); M(r); M(A)|] ==> wellfounded_on(M,A,r) <-> wf[A](r)"
by (blast intro: wellfounded_on_imp_wf_on wf_on_imp_relativized)

end
```

17 Separation for Facts About Order Types, Rank Functions and Well-Founded Relations

```
theory Rank_Separation imports Rank Rec_Separation begin
```

This theory proves all instances needed for locales *M_ordertype* and *M_wfrank*. But the material is not needed for proving the relative consistency of AC.

17.1 The Locale *M_ordertype*

17.1.1 Separation for Order-Isomorphisms

```
lemma well_ord_iso_Reflects:
  "REFLECTS[λx. x∈A -->
```

```
            (∃y[L]. ∃p[L]. fun_apply(L,f,x,y) & pair(L,y,x,p) & p
∈ r),
      λi x. x∈A --> (∃y ∈ Lset(i). ∃p ∈ Lset(i).
            fun_apply(##Lset(i),f,x,y) & pair(##Lset(i),y,x,p) & p
∈ r)]"
by (intro FOL_reflections function_reflections)

lemma well_ord_iso_separation:
    "[| L(A); L(f); L(r) |]
     ==> separation (L, λx. x∈A --> (∃y[L]. (∃p[L].
                 fun_apply(L,f,x,y) & pair(L,y,x,p) & p ∈ r)))"
apply (rule gen_separation_multi [OF well_ord_iso_Reflects, of "{A,f,r}"],

       auto)
apply (rule_tac env="[A,f,r]" in DPow_LsetI)
apply (rule sep_rules | simp)+
done
```

17.1.2 Separation for *obase*

```
lemma obase_reflects:
  "REFLECTS[λa. ∃x[L]. ∃g[L]. ∃mx[L]. ∃par[L].
          ordinal(L,x) & membership(L,x,mx) & pred_set(L,A,a,r,par)
&
          order_isomorphism(L,par,r,x,mx,g),
      λi a. ∃x ∈ Lset(i). ∃g ∈ Lset(i). ∃mx ∈ Lset(i). ∃par ∈ Lset(i).
          ordinal(##Lset(i),x) & membership(##Lset(i),x,mx) & pred_set(##Lset(i),A,a,r,p
&
          order_isomorphism(##Lset(i),par,r,x,mx,g)]"
by (intro FOL_reflections function_reflections fun_plus_reflections)

lemma obase_separation:
    — part of the order type formalization
    "[| L(A); L(r) |]
     ==> separation(L, λa. ∃x[L]. ∃g[L]. ∃mx[L]. ∃par[L].
          ordinal(L,x) & membership(L,x,mx) & pred_set(L,A,a,r,par)
&
          order_isomorphism(L,par,r,x,mx,g))"
apply (rule gen_separation_multi [OF obase_reflects, of "{A,r}"], auto)
apply (rule_tac env="[A,r]" in DPow_LsetI)
apply (rule ordinal_iff_sats sep_rules | simp)+
done
```

17.1.3 Separation for a Theorem about *obase*

```
lemma obase_equals_reflects:
  "REFLECTS[λx. x∈A --> ~(∃y[L]. ∃g[L].
            ordinal(L,y) & (∃my[L]. ∃pxr[L].
            membership(L,y,my) & pred_set(L,A,x,r,pxr) &
            order_isomorphism(L,pxr,r,y,my,g))),
```

```
        λi x. x∈A --> ~(∃y ∈ Lset(i). ∃g ∈ Lset(i).
                ordinal(##Lset(i),y) & (∃my ∈ Lset(i). ∃pxr ∈ Lset(i).
                membership(##Lset(i),y,my) & pred_set(##Lset(i),A,x,r,pxr)
&
                order_isomorphism(##Lset(i),pxr,r,y,my,g)))]"
```
by *(intro FOL_reflections function_reflections fun_plus_reflections)*

lemma *obase_equals_separation:*
```
     "[| L(A); L(r) |]
      ==> separation (L, λx. x∈A --> ~(∃y[L]. ∃g[L].
                             ordinal(L,y) & (∃my[L]. ∃pxr[L].
                             membership(L,y,my) & pred_set(L,A,x,r,pxr)
&
                             order_isomorphism(L,pxr,r,y,my,g))))"
```
apply *(rule gen_separation_multi [OF obase_equals_reflects, of "{A,r}"], auto)*
apply *(rule_tac env="[A,r]"* **in** *DPow_LsetI)*
apply *(rule sep_rules | simp)+*
done

17.1.4 Replacement for *omap*

lemma *omap_reflects:*
```
 "REFLECTS[λz. ∃a[L]. a∈B & (∃x[L]. ∃g[L]. ∃mx[L]. ∃par[L].
     ordinal(L,x) & pair(L,a,x,z) & membership(L,x,mx) &
     pred_set(L,A,a,r,par) & order_isomorphism(L,par,r,x,mx,g)),
 λi z. ∃a ∈ Lset(i). a∈B & (∃x ∈ Lset(i). ∃g ∈ Lset(i). ∃mx ∈ Lset(i).
        ∃par ∈ Lset(i).
         ordinal(##Lset(i),x) & pair(##Lset(i),a,x,z) &
         membership(##Lset(i),x,mx) & pred_set(##Lset(i),A,a,r,par) &
         order_isomorphism(##Lset(i),par,r,x,mx,g))]"
```
by *(intro FOL_reflections function_reflections fun_plus_reflections)*

lemma *omap_replacement:*
```
     "[| L(A); L(r) |]
      ==> strong_replacement(L,
             λa z. ∃x[L]. ∃g[L]. ∃mx[L]. ∃par[L].
             ordinal(L,x) & pair(L,a,x,z) & membership(L,x,mx) &
             pred_set(L,A,a,r,par) & order_isomorphism(L,par,r,x,mx,g))"
```
apply *(rule strong_replacementI)*
apply *(rule_tac u="{A,r,B}"* **in** *gen_separation_multi [OF omap_reflects], auto)*
apply *(rule_tac env="[A,B,r]"* **in** *DPow_LsetI)*
apply *(rule sep_rules | simp)+*
done

17.2 Instantiating the locale *M_ordertype*

Separation (and Strong Replacement) for basic set-theoretic constructions such as intersection, Cartesian Product and image.

```
lemma M_ordertype_axioms_L: "M_ordertype_axioms(L)"
  apply (rule M_ordertype_axioms.intro)
       apply (assumption | rule well_ord_iso_separation
         obase_separation obase_equals_separation
         omap_replacement)+
  done

theorem M_ordertype_L: "PROP M_ordertype(L)"
  apply (rule M_ordertype.intro)
   apply (rule M_basic_L)
  apply (rule M_ordertype_axioms_L)
  done
```

17.3 The Locale *M_wfrank*

17.3.1 Separation for *wfrank*

```
lemma wfrank_Reflects:
 "REFLECTS[λx. ∀rplus[L]. tran_closure(L,r,rplus) -->
              ~ (∃f[L]. M_is_recfun(L, %x f y. is_range(L,f,y), rplus,
x, f)),
      λi x. ∀rplus ∈ Lset(i). tran_closure(##Lset(i),r,rplus) -->
         ~ (∃f ∈ Lset(i).
            M_is_recfun(##Lset(i), %x f y. is_range(##Lset(i),f,y),
                        rplus, x, f))]"
by (intro FOL_reflections function_reflections is_recfun_reflection tran_closure_reflection

lemma wfrank_separation:
     "L(r) ==>
      separation (L, λx. ∀rplus[L]. tran_closure(L,r,rplus) -->
         ~ (∃f[L]. M_is_recfun(L, %x f y. is_range(L,f,y), rplus, x,
f)))"
apply (rule gen_separation [OF wfrank_Reflects], simp)
apply (rule_tac env="[r]" in DPow_LsetI)
apply (rule sep_rules tran_closure_iff_sats is_recfun_iff_sats | simp)+
done
```

17.3.2 Replacement for *wfrank*

```
lemma wfrank_replacement_Reflects:
 "REFLECTS[λz. ∃x[L]. x ∈ A &
        (∀rplus[L]. tran_closure(L,r,rplus) -->
         (∃y[L]. ∃f[L]. pair(L,x,y,z)  &
                        M_is_recfun(L, %x f y. is_range(L,f,y), rplus,
x, f) &
                        is_range(L,f,y))),
```

```
 λi z. ∃x ∈ Lset(i). x ∈ A &
      (∀rplus ∈ Lset(i). tran_closure(##Lset(i),r,rplus) -->
       (∃y ∈ Lset(i). ∃f ∈ Lset(i). pair(##Lset(i),x,y,z)  &
         M_is_recfun(##Lset(i), %x f y. is_range(##Lset(i),f,y), rplus,
x, f) &
         is_range(##Lset(i),f,y)))]"
by (intro FOL_reflections function_reflections fun_plus_reflections
            is_recfun_reflection tran_closure_reflection)
```

```
lemma wfrank_strong_replacement:
     "L(r) ==>
      strong_replacement(L, λx z.
         ∀rplus[L]. tran_closure(L,r,rplus) -->
         (∃y[L]. ∃f[L]. pair(L,x,y,z)  &
                        M_is_recfun(L, %x f y. is_range(L,f,y), rplus,
x, f) &
                        is_range(L,f,y)))"
apply (rule strong_replacementI)
apply (rule_tac u="{r,B}"
         in gen_separation_multi [OF wfrank_replacement_Reflects],
       auto)
apply (rule_tac env="[r,B]" in DPow_LsetI)
apply (rule sep_rules tran_closure_iff_sats is_recfun_iff_sats | simp)+
done
```

17.3.3 Separation for Proving `Ord_wfrank_range`

```
lemma Ord_wfrank_Reflects:
 "REFLECTS[λx. ∀rplus[L]. tran_closure(L,r,rplus) -->
          ~ (∀f[L]. ∀rangef[L].
             is_range(L,f,rangef) -->
             M_is_recfun(L, λx f y. is_range(L,f,y), rplus, x, f) -->
             ordinal(L,rangef)),
      λi x. ∀rplus ∈ Lset(i). tran_closure(##Lset(i),r,rplus) -->
          ~ (∀f ∈ Lset(i). ∀rangef ∈ Lset(i).
             is_range(##Lset(i),f,rangef) -->
             M_is_recfun(##Lset(i), λx f y. is_range(##Lset(i),f,y),
                         rplus, x, f) -->
             ordinal(##Lset(i),rangef))]"
by (intro FOL_reflections function_reflections is_recfun_reflection
          tran_closure_reflection ordinal_reflection)
```

```
lemma  Ord_wfrank_separation:
     "L(r) ==>
      separation (L, λx.
         ∀rplus[L]. tran_closure(L,r,rplus) -->
          ~ (∀f[L]. ∀rangef[L].
             is_range(L,f,rangef) -->
             M_is_recfun(L, λx f y. is_range(L,f,y), rplus, x, f) -->
```

```
             ordinal(L,rangef)))"
apply (rule gen_separation [OF Ord_wfrank_Reflects], simp)
apply (rule_tac env="[r]" in DPow_LsetI)
apply (rule sep_rules tran_closure_iff_sats is_recfun_iff_sats | simp)+
done
```

17.3.4 Instantiating the locale *M_wfrank*

```
lemma M_wfrank_axioms_L: "M_wfrank_axioms(L)"
  apply (rule M_wfrank_axioms.intro)
   apply (assumption | rule
     wfrank_separation wfrank_strong_replacement Ord_wfrank_separation)+
  done

theorem M_wfrank_L: "PROP M_wfrank(L)"
  apply (rule M_wfrank.intro)
   apply (rule M_trancl_L)
  apply (rule M_wfrank_axioms_L)
  done

lemmas exists_wfrank = M_wfrank.exists_wfrank [OF M_wfrank_L]
  and M_wellfoundedrank = M_wfrank.M_wellfoundedrank [OF M_wfrank_L]
  and Ord_wfrank_range = M_wfrank.Ord_wfrank_range [OF M_wfrank_L]
  and Ord_range_wellfoundedrank = M_wfrank.Ord_range_wellfoundedrank [OF
M_wfrank_L]
  and function_wellfoundedrank = M_wfrank.function_wellfoundedrank [OF
M_wfrank_L]
  and domain_wellfoundedrank = M_wfrank.domain_wellfoundedrank [OF M_wfrank_L]
  and wellfoundedrank_type = M_wfrank.wellfoundedrank_type [OF M_wfrank_L]
  and Ord_wellfoundedrank = M_wfrank.Ord_wellfoundedrank [OF M_wfrank_L]
  and wellfoundedrank_eq = M_wfrank.wellfoundedrank_eq [OF M_wfrank_L]
  and wellfoundedrank_lt = M_wfrank.wellfoundedrank_lt [OF M_wfrank_L]
  and wellfounded_imp_subset_rvimage = M_wfrank.wellfounded_imp_subset_rvimage
[OF M_wfrank_L]
  and wellfounded_imp_wf = M_wfrank.wellfounded_imp_wf [OF M_wfrank_L]
  and wellfounded_on_imp_wf_on = M_wfrank.wellfounded_on_imp_wf_on [OF
M_wfrank_L]
  and wf_abs = M_wfrank.wf_abs [OF M_wfrank_L]
  and wf_on_abs = M_wfrank.wf_on_abs [OF M_wfrank_L]

end
```

References

[1] Kurt Gödel. The consistency of the axiom of choice and of the generalized continuum hypothesis with the axioms of set theory. In S. Feferman et al., editors, *Kurt Gödel: Collected Works*, volume II. Oxford University Press, 1990.

[2] Kenneth Kunen. *Set Theory: An Introduction to Independence Proofs*. North-Holland, 1980.

[3] Lawrence C. Paulson. The reflection theorem: A study in meta-theoretic reasoning. In Andrei Voronkov, editor, *Automated Deduction — CADE-18 International Conference*, LNAI 2392, pages 377–391. Springer, 2002.

[4] Lawrence C. Paulson. The relative consistency of the axiom of choice — mechanized using Isabelle/ZF. *LMS Journal of Computation and Mathematics*, 6:198–248, 2003. http://www.lms.ac.uk/jcm/6/lms2003-001/.