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

Lawrence C Paulson

November 22, 2007

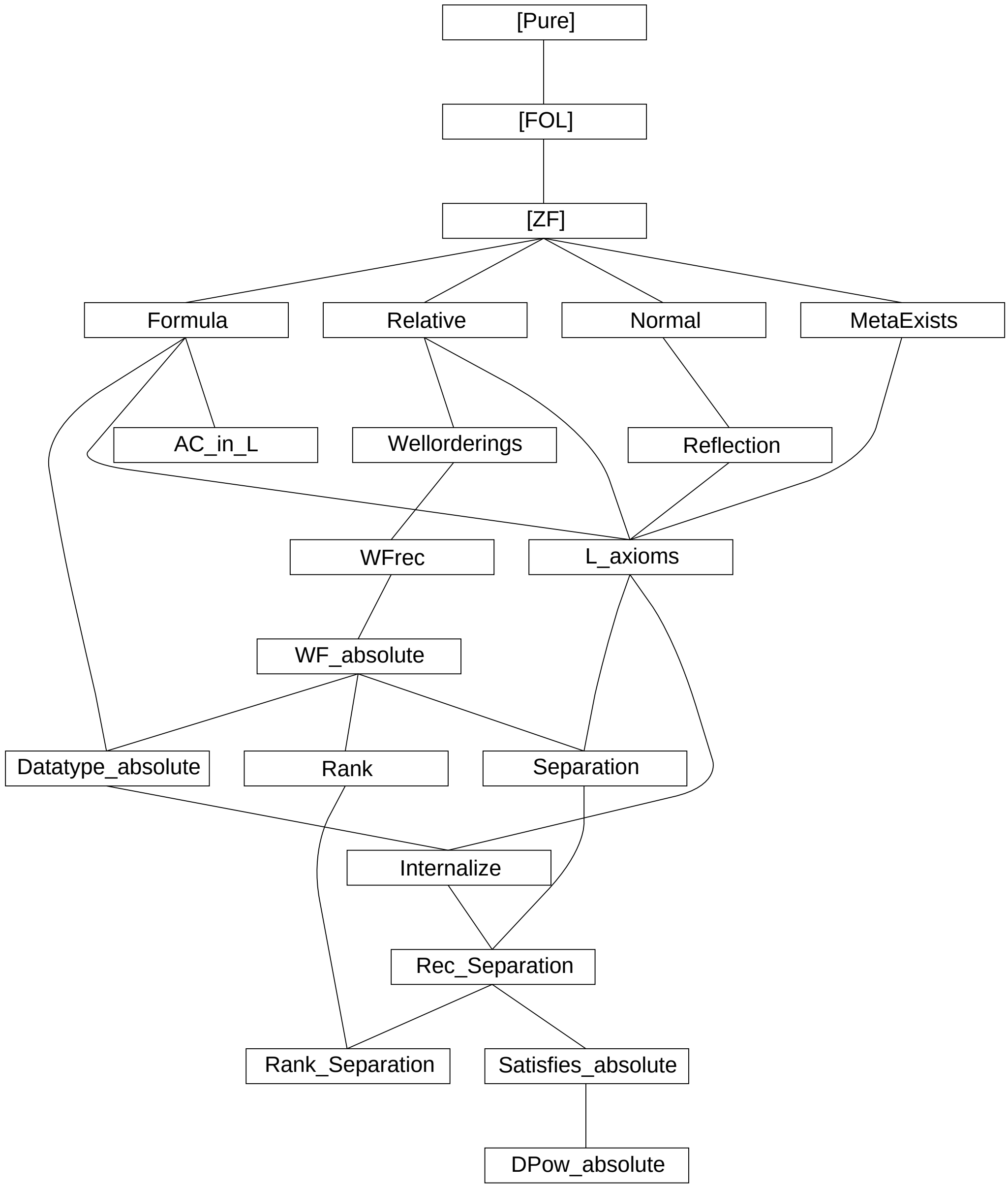
Abstract

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"
⟨proof⟩

lemma And_type [TC]: "[| p ∈ formula; q ∈ formula |] ==> And(p,q) ∈
formula"
⟨proof⟩
```

lemma *Or_type [TC]: "[| p ∈ formula; q ∈ formula |] ==> Or(p,q) ∈ formula"*
⟨*proof*⟩

lemma *Implies_type [TC]:*
"[| p ∈ formula; q ∈ formula |] ==> Implies(p,q) ∈ formula"
⟨*proof*⟩

lemma *Iff_type [TC]:*
"[| p ∈ formula; q ∈ formula |] ==> Iff(p,q) ∈ formula"
⟨*proof*⟩

lemma *Exists_type [TC]: "p ∈ formula ==> Exists(p) ∈ formula"*
⟨*proof*⟩

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"*
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *[simp]:*
"env ∈ list(A)
==> sats(A, Equal(x,y), env) <-> nth(x,env) = nth(y,env)"
⟨*proof*⟩

lemma *sats_Nand_iff [simp]:*
```
"env ∈ list(A)
 ==> (sats(A, Nand(p,q), env)) <-> ~ (sats(A,p,env) & sats(A,q,env))"
```

⟨*proof*⟩

lemma *sats_Forall_iff [simp]:*
```
"env ∈ list(A)
 ==> sats(A, Forall(p), env) <-> (∀x∈A. sats(A, p, Cons(x,env)))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *sats_And_iff [simp]:*
```
"env ∈ list(A)
 ==> (sats(A, And(p,q), env)) <-> sats(A,p,env) & sats(A,q,env)"
```
⟨*proof*⟩

lemma *sats_Or_iff [simp]:*
```
"env ∈ list(A)
 ==> (sats(A, Or(p,q), env)) <-> sats(A,p,env) | sats(A,q,env)"
```
⟨*proof*⟩

lemma *sats_Implies_iff [simp]:*
```
"env ∈ list(A)
 ==> (sats(A, Implies(p,q), env)) <-> (sats(A,p,env) --> sats(A,q,env))"
```
⟨*proof*⟩

lemma *sats_Iff_iff [simp]:*
```
"env ∈ list(A)
 ==> (sats(A, Iff(p,q), env)) <-> (sats(A,p,env) <-> sats(A,q,env))"
```
⟨*proof*⟩

lemma *sats_Exists_iff [simp]:*
```
"env ∈ list(A)
 ==> sats(A, Exists(p), env) <-> (∃x∈A. sats(A, p, Cons(x,env)))"
```
⟨*proof*⟩

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)"
```

⟨proof⟩

lemma *equal_iff_sats:*
```
"[| nth(i,env) = x; nth(j,env) = y; env ∈ list(A)|]
 ==> (x=y) <-> sats(A, Equal(i,j), env)"
```
⟨proof⟩

lemma *not_iff_sats:*
```
"[| P <-> sats(A,p,env); env ∈ list(A)|]
 ==> (~P) <-> sats(A, Neg(p), env)"
```
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨proof⟩

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"*
⟨*proof*⟩

lemma *arity_Neg [simp]: "arity(Neg(p)) = arity(p)"*
⟨*proof*⟩

lemma *arity_And [simp]: "arity(And(p,q)) = arity(p) ∪ arity(q)"*
⟨*proof*⟩

lemma *arity_Or [simp]: "arity(Or(p,q)) = arity(p) ∪ arity(q)"*
⟨*proof*⟩

lemma *arity_Implies [simp]: "arity(Implies(p,q)) = arity(p) ∪ arity(q)"*
⟨*proof*⟩

lemma *arity_Iff [simp]: "arity(Iff(p,q)) = arity(p) ∪ arity(q)"*
⟨*proof*⟩

lemma *arity_Exists [simp]: "arity(Exists(p)) = Arith.pred(arity(p))"*
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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"
⟨proof⟩

lemma incr_var_le: "nq≤x ==> incr_var(x,nq) = succ(x)"
⟨proof⟩

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"
⟨proof⟩

lemma incr_bv_type [TC]: "p ∈ formula ==> incr_bv(p) ∈ nat -> formula"
⟨proof⟩
```

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)"
⟨proof⟩



lemma incr_var_lemma:
     "[| x ∈ nat; y ∈ nat; nq ≤ x |]
      ==> succ(x) ∪ incr_var(y,nq) = succ(x ∪ y)"
⟨proof⟩
```

lemma *incr_And_lemma:*
```
    "y < x ==> y ∪ succ(x) = succ(x ∪ y)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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"*
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *arity_incr_bv1_eq:*
```
  "p ∈ formula
   ==> arity(incr_bv1(p)) =
        (if 1 < arity(p) then succ(arity(p)) else arity(p))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))})"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *DPow_subset_Pow: "DPow(A) <= Pow(A)"*
⟨*proof*⟩

lemma *empty_in_DPow: "0 ∈ DPow(A)"*
⟨*proof*⟩

lemma *Compl_in_DPow: "X ∈ DPow(A) ==> (A-X) ∈ DPow(A)"*
⟨*proof*⟩

lemma *Int_in_DPow: "[| X ∈ DPow(A); Y ∈ DPow(A) |] ==> X Int Y ∈ DPow(A)"*
⟨*proof*⟩

lemma *Un_in_DPow: "[| X ∈ DPow(A); Y ∈ DPow(A) |] ==> X Un Y ∈ DPow(A)"*
⟨*proof*⟩

lemma *singleton_in_DPow: "a ∈ A ==> {a} ∈ DPow(A)"*
⟨*proof*⟩

lemma *cons_in_DPow: "[| a ∈ A; X ∈ DPow(A) |] ==> cons(a,X) ∈ DPow(A)"*
⟨*proof*⟩

lemma *Fin_into_DPow: "X ∈ Fin(A) ==> X ∈ DPow(A)"*
⟨*proof*⟩

DPow is not monotonic. For example, let *A* be some non-constructible set of natural numbers, and let *B* be *nat*. Then *A ⊆ B* and obviously *A ∈ DPow(A)* but *A ∉ DPow(B)*.

lemma *Finite_Pow_subset_Pow: "Finite(A) ==> Pow(A) <= DPow(A)"*
⟨*proof*⟩

lemma *Finite_DPow_eq_Pow: "Finite(A) ==> DPow(A) = Pow(A)"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *arity_subset_fm [simp]:*
    "[| x ∈ nat; y ∈ nat |] ==> arity(subset_fm(x,y)) = succ(x) ∪ succ(y)"
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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"`
⟨*proof*⟩

lemma `arity_transset_fm [simp]:`
```
     "x ∈ nat ==> arity(transset_fm(x)) = succ(x)"
```
⟨*proof*⟩

lemma `sats_transset_fm [simp]:`
```
   "[|x < length(env); env ∈ list(A); Transset(A)|]
    ==> sats(A, transset_fm(x), env) <-> Transset(nth(x,env))"
```
⟨*proof*⟩

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"`
⟨*proof*⟩

lemma `arity_ordinal_fm [simp]:`
```
     "x ∈ nat ==> arity(ordinal_fm(x)) = succ(x)"
```
⟨*proof*⟩

lemma `sats_ordinal_fm:`
```
   "[|x < length(env); env ∈ list(A); Transset(A)|]
    ==> sats(A, ordinal_fm(x), env) <-> Ord(nth(x,env))"
```
⟨*proof*⟩

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)"`
⟨*proof*⟩

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)))"*
⟨*proof*⟩

lemma *LsetI: "[|y∈x; A ∈ DPow(Lset(y))|] ==> A ∈ Lset(x)"*
⟨*proof*⟩

lemma *LsetD: "A ∈ Lset(x) ==> ∃y∈x. A ∈ DPow(Lset(y))"*
⟨*proof*⟩

1.8.1 Transitivity

lemma *elem_subset_in_DPow: "[|X ∈ A; X ⊆ A|] ==> X ∈ DPow(A)"*
⟨*proof*⟩

lemma *Transset_subset_DPow: "Transset(A) ==> A <= DPow(A)"*
⟨*proof*⟩

lemma *Transset_DPow: "Transset(A) ==> Transset(DPow(A))"*
⟨*proof*⟩

Kunen's VI 1.6 (a)

lemma *Transset_Lset: "Transset(Lset(i))"*
⟨*proof*⟩

lemma *mem_Lset_imp_subset_Lset: "a ∈ Lset(i) ==> a ⊆ Lset(i)"*
⟨*proof*⟩

1.8.2 Monotonicity

Kunen's VI 1.6 (b)

lemma *Lset_mono [rule_format]:*
"ALL j. i<=j --> Lset(i) <= Lset(j)"
⟨*proof*⟩

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*⟩

Useful with Reflection to bump up the ordinal

lemma *subset_Lset_ltD: "[|A ⊆ Lset(i); i < j|] ==> A ⊆ Lset(j)"*
⟨*proof*⟩

1.8.3 0, successor and limit equations for Lset

lemma *Lset_0 [simp]: "Lset(0) = 0"*
⟨*proof*⟩

lemma *Lset_succ_subset1: "DPow(Lset(i)) <= Lset(succ(i))"*
⟨*proof*⟩

lemma *Lset_succ_subset2: "Lset(succ(i)) <= DPow(Lset(i))"*
⟨*proof*⟩

lemma *Lset_succ: "Lset(succ(i)) = DPow(Lset(i))"*
⟨*proof*⟩

lemma *Lset_Union [simp]: "Lset(⋃(X)) = (⋃y∈X. Lset(y))"*
⟨*proof*⟩

1.8.4 Lset applied to Limit ordinals

lemma *Limit_Lset_eq:*
```
    "Limit(i) ==> Lset(i) = (⋃y∈i. Lset(y))"
```
⟨*proof*⟩

lemma *lt_LsetI: "[| a: Lset(j); j<i |] ==> a ∈ Lset(i)"*
⟨*proof*⟩

lemma *Limit_LsetE:*
```
    "[| a: Lset(i);  ~R ==> Limit(i);
        !!x. [| x<i;  a: Lset(x) |] ==> R
     |] ==> R"
```
⟨*proof*⟩

1.8.5 Basic closure properties

lemma *zero_in_Lset: "y:x ==> 0 ∈ Lset(x)"*
⟨*proof*⟩

lemma *notin_Lset: "x ∉ Lset(x)"*
⟨*proof*⟩

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

lemma *Ords_of_Lset_eq: "Ord(i) ==> {x∈Lset(i). Ord(x)} = i"*
⟨*proof*⟩

lemma *Ord_subset_Lset: "Ord(i) ==> i ⊆ Lset(i)"*
⟨*proof*⟩

lemma *Ord_in_Lset: "Ord(i) ==> i ∈ Lset(succ(i))"*

⟨*proof*⟩

lemma *Ord_in_L: "Ord(i) ==> L(i)"*
⟨*proof*⟩

1.9.1 Unions

lemma *Union_in_Lset:*
    "X ∈ Lset(i) ==> Union(X) ∈ Lset(succ(i))"
⟨*proof*⟩

theorem *Union_in_L: "L(X) ==> L(Union(X))"*
⟨*proof*⟩

1.9.2 Finite sets and ordered pairs

lemma *singleton_in_Lset: "a: Lset(i) ==> {a} ∈ Lset(succ(i))"*
⟨*proof*⟩

lemma *doubleton_in_Lset:*
    "[| a: Lset(i); b: Lset(i) |] ==> {a,b} ∈ Lset(succ(i))"
⟨*proof*⟩

lemma *Pair_in_Lset:*
    "[| a: Lset(i); b: Lset(i); Ord(i) |] ==> <a,b> ∈ Lset(succ(succ(i)))"
⟨*proof*⟩

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)"
⟨*proof*⟩

theorem *doubleton_in_L: "[| L(a); L(b) |] ==> L({a, b})"*
⟨*proof*⟩

lemma *Pair_in_LLimit:*
    "[| a: Lset(i); b: Lset(i); Limit(i) |] ==> <a,b> ∈ Lset(i)"⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *L_D: "L(x) ==> ∃i. Ord(i) & x ∈ Lset(i)"*
⟨*proof*⟩

lemma *Ord_lrank [simp]: "Ord(lrank(a))"*
⟨*proof*⟩

lemma *Lset_lrank_lt [rule_format]: "Ord(i) ==> x ∈ Lset(i) --> lrank(x) < i"*
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *Lset_succ_lrank_iff [simp]: "x ∈ Lset(succ(lrank(x))) <-> L(x)"*
⟨*proof*⟩

Kunen's VI 1.9 (a)

lemma *lrank_of_Ord: "Ord(i) ==> lrank(i) = i"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *lrank_Lset: "Ord(i) ==> lrank(Lset(i)) = i"*
⟨*proof*⟩

Kunen's VI 1.11

lemma *Lset_subset_Vset: "Ord(i) ==> Lset(i) <= Vset(i)"*
⟨*proof*⟩

Kunen's VI 1.12

lemma *Lset_subset_Vset': "i ∈ nat ==> Lset(i) = Vset(i)"*
⟨*proof*⟩

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

lemma *subset_Lset:*
    "(∀x∈A. L(x)) ==> ∃i. Ord(i) & A ⊆ Lset(i)"
⟨*proof*⟩

lemma *subset_LsetE:*

```
  "[|∀x∈A. L(x);
     !!i. [|Ord(i); A ⊆ Lset(i)|] ==> P|]
   ==> P"
```
⟨*proof*⟩

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)))"
```
⟨*proof*⟩

lemma *LPow_in_Lset:*
```
  "[|X ∈ Lset(i); Ord(i)|] ==> ∃j. Ord(j) & {y ∈ Pow(X). L(y)} ∈
Lset(j)"
```
⟨*proof*⟩

theorem *LPow_in_L: "L(X) ==> L({y ∈ Pow(X). L(y)})"*
⟨*proof*⟩

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

lemma *nth_zero_eq_0: "n ∈ nat ==> nth(n,[0]) = 0"*
⟨*proof*⟩

lemma *sats_app_0_iff [rule_format]:*
```
  "[| p ∈ formula; 0 ∈ A |]
   ==> ∀env ∈ list(A). sats(A,p, env@[0]) <-> sats(A,p,env)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)))"
```
⟨*proof*⟩

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)"*

⟨proof⟩

lemma *DPow'_0: "DPow'(0) = {0}"*
⟨proof⟩

lemma *DPow'_subset_DPow: "0 ∈ A ==> DPow'(A) ⊆ DPow(A)"*
⟨proof⟩

lemma *DPow_eq_DPow': "Transset(A) ==> DPow(A) = DPow'(A)"*
⟨proof⟩

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))"*
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨*proof*⟩

lemma *triv_Relation2:*
```
     "Relation2(M, A, B, λx y a. a = f(x,y), f)"
```
⟨*proof*⟩

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)"`
⟨*proof*⟩

lemma `univ0_Ball_abs [simp]:`
```
     "A ∈ univ(0) ==> (∀x∈A. x ∈ univ(0) --> P(x)) <-> (∀x∈A. P(x))"
```
⟨*proof*⟩

lemma `univ0_Bex_abs [simp]:`
```
     "A ∈ univ(0) ==> (∃x∈A. x ∈ univ(0) & P(x)) <-> (∃x∈A. P(x))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma `univalent_triv [intro,simp]:`
```
     "univalent(M, A, λx y. y = f(x))"
```
⟨*proof*⟩

lemma `univalent_conjI2 [intro,simp]:`
```
     "univalent(M,A,Q) ==> univalent(M, A, λx y. P(x,y) & Q(x,y))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

The extensionality axiom

lemma *"extensionality(λx. x ∈ univ(0))"*
⟨*proof*⟩

The separation axiom requires some lemmas

lemma *Collect_in_Vfrom:*
```
"[| X ∈ Vfrom(A,j);  Transset(A) |] ==> Collect(X,P) ∈ Vfrom(A,
succ(j))"
```
⟨*proof*⟩

lemma *Collect_in_VLimit:*
```
"[| X ∈ Vfrom(A,i);  Limit(i);  Transset(A) |]
 ==> Collect(X,P) ∈ Vfrom(A,i)"
```
⟨*proof*⟩

lemma *Collect_in_univ:*
```
"[| X ∈ univ(A);  Transset(A) |] ==> Collect(X,P) ∈ univ(A)"
```
⟨*proof*⟩

lemma *"separation(λx. x ∈ univ(0), P)"*
⟨*proof*⟩

Unordered pairing axiom

lemma *"upair_ax(λx. x ∈ univ(0))"*
⟨*proof*⟩

Union axiom

lemma *"Union_ax(λx. x ∈ univ(0))"*
⟨*proof*⟩

Powerset axiom

lemma *Pow_in_univ:*
```
"[| X ∈ univ(A);  Transset(A) |] ==> Pow(X) ∈ univ(A)"
```
⟨*proof*⟩

lemma *"power_ax(λx. x ∈ univ(0))"*
⟨*proof*⟩

Foundation axiom

lemma *"foundation_ax(λx. x ∈ univ(0))"*
⟨*proof*⟩

lemma *"replacement(λx. x ∈ univ(0), P)"*
⟨*proof*⟩

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>}"*
⟨*proof*⟩

lemma *vimage_iff_Collect:*
```
"r -`` A = {x ∈ Union(Union(r)). ∃p∈r. ∃y∈A. p=<x,y>}"
```
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *range_eq_image: "range(r) = r `` Union(Union(r))"*
⟨*proof*⟩

lemma *replacementD:*
```
"[| replacement(M,P); M(A);  univalent(M,A,P) |]
 ==> ∃Y[M]. (∀b[M]. ((∃x[M]. x∈A & P(x,b)) --> b ∈ Y))"
```
⟨*proof*⟩

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))))"
```
⟨*proof*⟩

lemma *separationD:*
```
"[| separation(M,P); M(z) |] ==> ∃y[M]. ∀x[M]. x ∈ y <-> x ∈ z &
P(x)"
```
⟨*proof*⟩

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)"
⟨proof⟩

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

lemma (in M_trivial) rex_abs [simp]:
     "M(A) ==> (∃x[M]. x∈A & P(x)) <-> (∃x∈A. P(x))"
⟨proof⟩

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)"
⟨proof⟩
```

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"
⟨*proof*⟩

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

lemma *(**in** M_trivial) empty_abs [simp]:*
"M(z) ==> empty(M,z) <-> z=0"
⟨*proof*⟩

lemma *(**in** M_trivial) subset_abs [simp]:*
"M(A) ==> subset(M,A,B) <-> A ⊆ B"
⟨*proof*⟩

lemma *(**in** M_trivial) upair_abs [simp]:*
"M(z) ==> upair(M,a,b,z) <-> z={a,b}"
⟨*proof*⟩

lemma *(**in** M_trivial) upair_in_M_iff [iff]:*
"M({a,b}) <-> M(a) & M(b)"
⟨*proof*⟩

lemma *(**in** M_trivial) singleton_in_M_iff [iff]:*
"M({a}) <-> M(a)"
⟨*proof*⟩

lemma *(**in** M_trivial) pair_abs [simp]:*
"M(z) ==> pair(M,a,b,z) <-> z=<a,b>"
⟨*proof*⟩

lemma *(**in** M_trivial) pair_in_M_iff [iff]:*
"M(<a,b>) <-> M(a) & M(b)"
⟨*proof*⟩

lemma *(**in** M_trivial) pair_components_in_M:*
"[| <x,y> ∈ A; M(A) |] ==> M(x) & M(y)"
⟨*proof*⟩

lemma *(**in** M_trivial) cartprod_abs [simp]:*
"[| M(A); M(B); M(z) |] ==> cartprod(M,A,B,z) <-> z = A*B"
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *(**in** M_trivial) inter_abs [simp]:*
"[| M(a); M(b); M(z) |] ==> inter(M,a,b,z) <-> z = a Int b"
⟨*proof*⟩

lemma `(in M_trivial) setdiff_abs [simp]:`
    `"[| M(a); M(b); M(z) |] ==> setdiff(M,a,b,z) <-> z = a-b"`
⟨*proof*⟩

lemma `(in M_trivial) Union_abs [simp]:`
    `"[| M(A); M(z) |] ==> big_union(M,A,z) <-> z = Union(A)"`
⟨*proof*⟩

lemma `(in M_trivial) Union_closed [intro,simp]:`
    `"M(A) ==> M(Union(A))"`
⟨*proof*⟩

lemma `(in M_trivial) Un_closed [intro,simp]:`
    `"[| M(A); M(B) |] ==> M(A Un B)"`
⟨*proof*⟩

lemma `(in M_trivial) cons_closed [intro,simp]:`
    `"[| M(a); M(A) |] ==> M(cons(a,A))"`
⟨*proof*⟩

lemma `(in M_trivial) cons_abs [simp]:`
    `"[| M(b); M(z) |] ==> is_cons(M,a,b,z) <-> z = cons(a,b)"`
⟨*proof*⟩

lemma `(in M_trivial) successor_abs [simp]:`
    `"[| M(a); M(z) |] ==> successor(M,a,z) <-> z = succ(a)"`
⟨*proof*⟩

lemma `(in M_trivial) succ_in_M_iff [iff]:`
    `"M(succ(a)) <-> M(a)"`
⟨*proof*⟩

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))"`
⟨*proof*⟩

lemma `separation_iff:`
    `"separation(M,P) <-> (∀z[M]. ∃y[M]. is_Collect(M,z,P,y))"`
⟨*proof*⟩

lemma `(in M_trivial) Collect_abs [simp]:`
    `"[| M(A); M(z) |] ==> is_Collect(M,A,P,z) <-> z = Collect(A,P)"`
⟨*proof*⟩

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)"
```
⟨proof⟩

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')"
```
⟨proof⟩

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))"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))"
```
⟨proof⟩

lemma *Replace_conj_eq: "{y . x ∈ A, x∈A & y=f(x)} = {y . x∈A, y=f(x)}"*
⟨proof⟩

Better than *RepFun_closed* when having the formula *x ∈ 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)))"
```
⟨proof⟩

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))"
```
⟨*proof*⟩

Better than `lam_closed`: has the formula `x ∈ 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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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')"
```
⟨*proof*⟩

lemma *(in M_trivial) image_abs [simp]:*
```
     "[| M(r); M(A); M(z) |] ==> image(M,r,A,z) <-> z = r``A"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

2.5.6 Absoluteness for the Natural Numbers

lemma *(in M_trivial) nat_into_M [intro]:*

```
     "n ∈ nat ==> M(n)"
```
⟨*proof*⟩

lemma *(in M_trivial) nat_case_closed [intro,simp]:*
```
  "[|M(k); M(a); ∀m[M]. M(b(m))|] ==> M(nat_case(a,b,k))"
```
⟨*proof*⟩

lemma *(in M_trivial) quasinat_abs [simp]:*
```
     "M(z) ==> is_quasinat(M,z) <-> quasinat(z)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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')"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *(in M_trivial) transitive_set_abs [simp]:*
```
     "M(a) ==> transitive_set(M,a) <-> Transset(a)"
```
⟨*proof*⟩

lemma *(in M_trivial) ordinal_abs [simp]:*
```
     "M(a) ==> ordinal(M,a) <-> Ord(a)"
```
⟨*proof*⟩

lemma *(in M_trivial) limit_ordinal_abs [simp]:*
```
     "M(a) ==> limit_ordinal(M,a) <-> Limit(a)"
```
⟨*proof*⟩

lemma *(in M_trivial) successor_ordinal_abs [simp]:*
```
     "M(a) ==> successor_ordinal(M,a) <-> Ord(a) & (∃b[M]. a = succ(b))"
```
⟨*proof*⟩

lemma *finite_Ord_is_nat:*
```
      "[| Ord(a); ~ Limit(a); ∀x∈a. ~ Limit(x) |] ==> a ∈ nat"
```

⟨*proof*⟩

lemma *(in M_trivial) finite_ordinal_abs [simp]:*
```
"M(a) ==> finite_ordinal(M,a) <-> a ∈ nat"
```
⟨*proof*⟩

lemma *Limit_non_Limit_implies_nat:*
```
"[| Limit(a); ∀x∈a. ~ Limit(x) |] ==> a = nat"
```
⟨*proof*⟩

lemma *(in M_trivial) omega_abs [simp]:*
```
"M(a) ==> omega(M,a) <-> a = nat"
```
⟨*proof*⟩

lemma *(in M_trivial) number1_abs [simp]:*
```
"M(a) ==> number1(M,a) <-> a = 1"
```
⟨*proof*⟩

lemma *(in M_trivial) number2_abs [simp]:*
```
"M(a) ==> number2(M,a) <-> a = succ(1)"
```
⟨*proof*⟩

lemma *(in M_trivial) number3_abs [simp]:*
```
"M(a) ==> number3(M,a) <-> a = succ(succ(1))"
```
⟨*proof*⟩

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}}}"
⟨proof⟩
```

```
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>})"
⟨proof⟩
```

```
lemma (in M_basic) cartprod_closed_lemma:
     "[| M(A); M(B) |] ==> ∃C[M]. cartprod(M,A,B,C)"
⟨proof⟩
```

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)"
⟨proof⟩
```

```
lemma (in M_basic) sum_closed [intro,simp]:
     "[| M(A); M(B) |] ==> M(A+B)"
⟨proof⟩
```

lemma *(in M_basic) sum_abs [simp]:*
"[| M(A); M(B); M(Z) |] ==> is_sum(M,A,B,Z) <-> (Z = A+B)"
⟨*proof*⟩

lemma *(in M_trivial) Inl_in_M_iff [iff]:*
"M(Inl(a)) <-> M(a)"
⟨*proof*⟩

lemma *(in M_trivial) Inl_abs [simp]:*
"M(Z) ==> is_Inl(M,a,Z) <-> (Z = Inl(a))"
⟨*proof*⟩

lemma *(in M_trivial) Inr_in_M_iff [iff]:*
"M(Inr(a)) <-> M(a)"
⟨*proof*⟩

lemma *(in M_trivial) Inr_abs [simp]:*
"M(Z) ==> is_Inr(M,a,Z) <-> (Z = Inr(a))"
⟨*proof*⟩

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⟩}"
⟨*proof*⟩

lemma *(in M_basic) converse_closed [intro,simp]:*
"M(r) ==> M(converse(r))"
⟨*proof*⟩

lemma *(in M_basic) converse_abs [simp]:*
"[| M(r); M(z) |] ==> is_converse(M,r,z) <-> z = converse(r)"
⟨*proof*⟩

2.7.2 image, preimage, domain, range

lemma *(in M_basic) image_closed [intro,simp]:*
"[| M(A); M(r) |] ==> M(r``A)"
⟨*proof*⟩

lemma *(in M_basic) vimage_abs [simp]:*
"[| M(r); M(A); M(z) |] ==> pre_image(M,r,A,z) <-> z = r-``A"
⟨*proof*⟩

lemma *(in M_basic) vimage_closed [intro,simp]:*
"[| M(A); M(r) |] ==> M(r-``A)"

⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *(in M_basic) domain_closed [intro,simp]:*
```
"M(r) ==> M(domain(r))"
```
⟨*proof*⟩

lemma *(in M_basic) range_abs [simp]:*
```
"[| M(r); M(z) |] ==> is_range(M,r,z) <-> z = range(r)"
```
⟨*proof*⟩

lemma *(in M_basic) range_closed [intro,simp]:*
```
"M(r) ==> M(range(r))"
```
⟨*proof*⟩

lemma *(in M_basic) field_abs [simp]:*
```
"[| M(r); M(z) |] ==> is_field(M,r,z) <-> z = field(r)"
```
⟨*proof*⟩

lemma *(in M_basic) field_closed [intro,simp]:*
```
"M(r) ==> M(field(r))"
```
⟨*proof*⟩

2.7.4 Relations, functions and application

lemma *(in M_basic) relation_abs [simp]:*
```
"M(r) ==> is_relation(M,r) <-> relation(r)"
```
⟨*proof*⟩

lemma *(in M_basic) function_abs [simp]:*
```
"M(r) ==> is_function(M,r) <-> function(r)"
```
⟨*proof*⟩

lemma *(in M_basic) apply_closed [intro,simp]:*
```
"[|M(f); M(a)|] ==> M(f`a)"
```
⟨*proof*⟩

lemma *(in M_basic) apply_abs [simp]:*
```
"[| M(f); M(x); M(y) |] ==> fun_apply(M,f,x,y) <-> f`x = y"
```
⟨*proof*⟩

lemma *(in M_basic) typed_function_abs [simp]:*
```
"[| M(A); M(f) |] ==> typed_function(M,A,B,f) <-> f ∈ A -> B"
```
⟨*proof*⟩

lemma *(in M_basic) injection_abs [simp]:*
```
"[| M(A); M(f) |] ==> injection(M,A,B,f) <-> f ∈ inj(A,B)"
```
⟨*proof*⟩

lemma *(in M_basic) surjection_abs [simp]:*
```
"[| M(A); M(B); M(f) |] ==> surjection(M,A,B,f) <-> f ∈ surj(A,B)"
```
⟨*proof*⟩

lemma *(in M_basic) bijection_abs [simp]:*
```
"[| M(A); M(B); M(f) |] ==> bijection(M,A,B,f) <-> f ∈ bij(A,B)"
```
⟨*proof*⟩

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}"
```
⟨*proof*⟩

lemma *(in M_basic) comp_closed [intro,simp]:*
```
"[| M(r); M(s) |] ==> M(r O s)"
```
⟨*proof*⟩

lemma *(in M_basic) composition_abs [simp]:*
```
"[| M(r); M(s); M(t) |] ==> composition(M,r,s,t) <-> t = r O s"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *(in M_basic) restriction_abs [simp]:*
```
"[| M(f); M(A); M(z) |]
 ==> restriction(M,f,A,z) <-> z = restrict(f,A)"
```
⟨*proof*⟩

lemma *(in M_basic) M_restrict_iff:*
```
"M(r) ==> restrict(r,A) = {z ∈ r . ∃x∈A. ∃y[M]. z = ⟨x, y⟩}"
```
⟨*proof*⟩

lemma *(in M_basic) restrict_closed [intro,simp]:*
```
"[| M(A); M(r) |] ==> M(restrict(r,A))"
```
⟨*proof*⟩

lemma *(in M_basic) Inter_abs [simp]:*
"[| M(A); M(z) |] ==> big_inter(M,A,z) <-> z = Inter(A)"
⟨*proof*⟩

lemma *(in M_basic) Inter_closed [intro,simp]:*
"M(A) ==> M(Inter(A))"
⟨*proof*⟩

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

lemma *(in M_basic) Diff_closed [intro,simp]:*
"[|M(A); M(B)|] ==> M(A-B)"
⟨*proof*⟩

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))"
⟨*proof*⟩

lemma *Collect_Un_Collect_eq:*
"Collect(A,P) Un Collect(A,Q) = Collect(A, %x. P(x) | Q(x))"
⟨*proof*⟩

lemma *Diff_Collect_eq:*
"A - Collect(A,P) = Collect(A, %x. ~ P(x))"
⟨*proof*⟩

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)}))"
⟨*proof*⟩

lemma *(in M_basic) separation_disj:*
"[|separation(M,P); separation(M,Q)|] ==> separation(M, λz. P(z) | Q(z))"
⟨*proof*⟩

lemma *(in M_basic) separation_neg:*
"separation(M,P) ==> separation(M, λz. ~P(z))"
⟨*proof*⟩

lemma *(in M_basic) separation_imp:*
"[|separation(M,P); separation(M,Q)|]
==> separation(M, λz. P(z) --> Q(z))"

⟨*proof*⟩

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))"
```
⟨*proof*⟩

2.7.7 Functions and function space

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

```
lemma (in M_basic) is_funspace_abs [simp]:
     "[|M(A); M(B); M(F); M(A->B)|] ==> is_funspace(M,A,B,F) <-> F = A->B"
```
⟨*proof*⟩

```
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)}}"
```
⟨*proof*⟩

```
lemma (in M_basic) funspace_succ:
     "[|M(n); M(B); M(n->B) |] ==> M(succ(n) -> B)"
```
⟨*proof*⟩

`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)"
```
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *(in M_trivial) not_abs [simp]:*
    "[| M(a); M(z)|] ==> is_not(M,a,z) <-> z = not(a)"
⟨*proof*⟩

lemma *(in M_trivial) and_abs [simp]:*
    "[| M(a); M(b); M(z)|] ==> is_and(M,a,b,z) <-> z = a and b"
⟨*proof*⟩

lemma *(in M_trivial) or_abs [simp]:*
    "[| M(a); M(b); M(z)|] ==> is_or(M,a,b,z) <-> z = a or b"
⟨*proof*⟩

lemma *(in M_trivial) bool_of_o_closed [intro,simp]:*
    "M(bool_of_o(P))"
⟨*proof*⟩

lemma *(in M_trivial) and_closed [intro,simp]:*
    "[| M(p); M(q) |] ==> M(p and q)"
⟨*proof*⟩

lemma *(in M_trivial) or_closed [intro,simp]:*
    "[| M(p); M(q) |] ==> M(p or q)"
⟨*proof*⟩

lemma *(in M_trivial) not_closed [intro,simp]:*
    "M(p) ==> M(not(p))"
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma (**in** *M_trivial*) *Nil_abs [simp]*: "M(Z) ==> is_Nil(M,Z) <-> (Z = Nil)"
⟨*proof*⟩

lemma (**in** *M_trivial*) *Cons_in_M_iff [iff]*: "M(Cons(a,l)) <-> M(a) & M(l)"
⟨*proof*⟩

lemma (**in** *M_trivial*) *Cons_abs [simp]*:
"[|M(a); M(l); M(Z)|] ==> is_Cons(M,a,l,Z) <-> (Z = Cons(a,l))"
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *[iff]: "quasilist(Cons(x,l))"*
⟨*proof*⟩

lemma *list_imp_quasilist: "l ∈ list(A) ==> quasilist(l)"*
⟨*proof*⟩

2.9.2 *list_case'*, the Modified Version of *list_case*

lemma *list_case'_Nil [simp]: "list_case'(a,b,Nil) = a"*
⟨*proof*⟩

lemma *list_case'_Cons [simp]: "list_case'(a,b,Cons(x,l)) = b(x,l)"*
⟨*proof*⟩

lemma *non_list_case: "~ quasilist(x) ==> list_case'(a,b,x) = 0"*
⟨*proof*⟩

lemma *list_case'_eq_list_case [simp]:*
"xs ∈ list(A) ==>list_case'(a,b,xs) = list_case(a,b,xs)"
⟨*proof*⟩

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))"
⟨*proof*⟩

lemma *(in M_trivial) quasilist_abs [simp]:*
"M(z) ==> is_quasilist(M,z) <-> quasilist(z)"
⟨*proof*⟩

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)"
⟨*proof*⟩

2.9.3 The Modified Operators *hd'* and *tl'*

lemma *(in M_trivial) is_hd_Nil: "is_hd(M,[],Z) <-> empty(M,Z)"*
⟨*proof*⟩

lemma *(in M_trivial) is_hd_Cons:*
"[|M(a); M(l)|] ==> is_hd(M,Cons(a,l),Z) <-> Z = a"
⟨*proof*⟩

lemma *(in M_trivial) hd_abs [simp]:*
"[|M(x); M(y)|] ==> is_hd(M,x,y) <-> y = hd'(x)"
⟨*proof*⟩

lemma *(in M_trivial) is_tl_Nil: "is_tl(M,[],Z) <-> Z = []"*
⟨*proof*⟩

lemma *(in M_trivial) is_tl_Cons:*
"[|M(a); M(l)|] ==> is_tl(M,Cons(a,l),Z) <-> Z = l"
⟨*proof*⟩

lemma *(in M_trivial) tl_abs [simp]:*
"[|M(x); M(y)|] ==> is_tl(M,x,y) <-> y = tl'(x)"
⟨*proof*⟩

lemma *(in M_trivial) relation1_tl: "relation1(M, is_tl(M), tl')"*
⟨*proof*⟩

lemma *hd'_Nil: "hd'([]) = 0"*
⟨*proof*⟩

lemma *hd'_Cons: "hd'(Cons(a,l)) = a"*
⟨*proof*⟩

lemma *tl'_Nil: "tl'([]) = []"*
⟨*proof*⟩

lemma *tl'_Cons: "tl'(Cons(a,l)) = l"*
⟨*proof*⟩

lemma *iterates_tl_Nil: "n ∈ nat ==> tl'^n ([]) = []"*
⟨*proof*⟩

lemma *(***in** *M_basic) tl'_closed: "M(x) ==> M(tl'(x))"*
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *(***in** *M_basic) transitive_rel_abs [simp]:*
     "M(A) ==> transitive_rel(M,A,r) <-> trans[A](r)"
⟨*proof*⟩

lemma *(***in** *M_basic) linear_rel_abs [simp]:*
     "M(A) ==> linear_rel(M,A,r) <-> linear(A,r)"
⟨*proof*⟩

lemma *(***in** *M_basic) wellordered_is_trans_on:*
    "[| wellordered(M,A,r); M(A) |] ==> trans[A](r)"
⟨*proof*⟩

lemma *(***in** *M_basic) wellordered_is_linear:*
    "[| wellordered(M,A,r); M(A) |] ==> linear(A,r)"
⟨*proof*⟩

lemma *(***in** *M_basic) wellordered_is_wellfounded_on:*
    "[| wellordered(M,A,r); M(A) |] ==> wellfounded_on(M,A,r)"
⟨*proof*⟩

lemma *(***in** *M_basic) wellfounded_imp_wellfounded_on:*
    "[| wellfounded(M,r); M(A) |] ==> wellfounded_on(M,A,r)"
⟨*proof*⟩

lemma *(***in** *M_basic) wellfounded_on_subset_A:*
     "[| wellfounded_on(M,A,r); B<=A |] ==> wellfounded_on(M,B,r)"
⟨*proof*⟩

3.1.2 Well-founded relations

lemma *(***in** *M_basic) wellfounded_on_iff_wellfounded:*
     *"wellfounded_on(M,A,r) <-> wellfounded(M, r ∩ A*A)"*
⟨*proof*⟩

lemma *(***in** *M_basic) wellfounded_on_imp_wellfounded:*
     *"[|wellfounded_on(M,A,r); r ⊆ A*A|] ==> wellfounded(M,r)"*
⟨*proof*⟩

lemma *(in M_basic) wellfounded_on_field_imp_wellfounded:*
"wellfounded_on(M, field(r), r) ==> wellfounded(M,r)"
⟨*proof*⟩

lemma *(in M_basic) wellfounded_iff_wellfounded_on_field:*
"M(r) ==> wellfounded(M,r) <-> wellfounded_on(M, field(r), r)"
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *(in M_basic) trans_on_imp_relativized:*
"trans[A](r) ==> transitive_rel(M,A,r)"
⟨*proof*⟩

lemma *(in M_basic) wf_on_imp_relativized:*
"wf[A](r) ==> wellfounded_on(M,A,r)"
⟨*proof*⟩

lemma *(in M_basic) wf_imp_relativized:*
"wf(r) ==> wellfounded(M,r)"
⟨*proof*⟩

lemma *(in M_basic) well_ord_imp_relativized:*
"well_ord(A,r) ==> wellordered(M,A,r)"
⟨*proof*⟩

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)"
```

⟨proof⟩

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)"
```
⟨proof⟩

lemma *(in M_basic) pred_closed [intro,simp]:*
```
"[| M(A); M(r); M(x) |] ==> M(Order.pred(A,x,r))"
```
⟨proof⟩

lemma *(in M_basic) membership_abs [simp]:*
```
"[| M(r); M(A) |] ==> membership(M,A,r) <-> r = Memrel(A)"
```
⟨proof⟩

lemma *(in M_basic) M_Memrel_iff:*
```
"M(A) ==>
 Memrel(A) = {z ∈ A*A. ∃x[M]. ∃y[M]. z = ⟨x,y⟩ & x ∈ y}"
```
⟨proof⟩

lemma *(in M_basic) Memrel_closed [intro,simp]:*
```
"M(A) ==> M(Memrel(A))"
```
⟨proof⟩

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)"
```
⟨proof⟩

lemma *transitive_rel_subset:*
```
"[| transitive_rel(M,A,r);  B<=A |] ==> transitive_rel(M,B,r)"
```
⟨proof⟩

lemma *wellfounded_on_subset:*
```
"[| wellfounded_on(M,A,r);  B<=A |] ==> wellfounded_on(M,B,r)"
```
⟨proof⟩

lemma *wellordered_subset:*
```
"[| wellordered(M,A,r);  B<=A |] ==> wellordered(M,B,r)"
```
⟨proof⟩

lemma *(in M_basic) wellfounded_on_asym:*
```
"[| wellfounded_on(M,A,r);  <a,x>∈r;  a∈A; x∈A;  M(A) |] ==> <x,a>∉r"
```
⟨proof⟩

lemma *(in M_basic) wellordered_asym:*
```
"[| wellordered(M,A,r);  <a,x>∈r;  a∈A; x∈A;  M(A) |] ==> <x,a>∉r"
```
⟨proof⟩

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}))"
```
⟨*proof*⟩

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})))"
```
⟨*proof*⟩

lemma *is_recfun_imp_in_r:* `"[|is_recfun(r,a,H,f); ⟨x,i⟩ ∈ f|] ==> ⟨x, a⟩ ∈ r"`
⟨*proof*⟩

lemma *trans_Int_eq:*
```
      "[| trans(r); <y,x> ∈ r |] ==> r -`` {x} ∩ r -`` {y} = r -`` {y}"
```
⟨*proof*⟩

lemma *is_recfun_restrict_idem:*
```
     "is_recfun(r,a,H,f) ==> restrict(f, r -`` {a}) = f"
```
⟨*proof*⟩

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')"
```
⟨*proof*⟩

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')"
```
⟨*proof*⟩

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))"
⟨proof⟩
```

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"
⟨proof⟩
```

```
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"
⟨proof⟩
```

```
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"
⟨proof⟩
```

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}))>))"
⟨proof⟩
```

```
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}))"
⟨proof⟩
```

```
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"
```

⟨*proof*⟩

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)))"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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')"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)))"
```
⟨proof⟩

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')"
```
⟨proof⟩

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^*"
```
⟨proof⟩

lemma *rtrancl_alt_subset_rtrancl: "rtrancl_alt(field(r),r) <= r^*"*
⟨proof⟩

lemma *rtrancl_subset_rtrancl_alt: "r^* <= rtrancl_alt(field(r),r)"*
⟨proof⟩

lemma *rtrancl_alt_eq_rtrancl: "rtrancl_alt(field(r),r) = r^*"*
⟨proof⟩

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)))"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

lemma *(in M_trancl) rtrancl_closed [intro,simp]:*

"M(r) ==> M(rtrancl(r))"
⟨proof⟩

lemma *(in* M_trancl*)* rtrancl_abs [simp]:
"[| M(r); M(z) |] ==> rtran_closure(M,r,z) <-> z = rtrancl(r)"
⟨proof⟩

lemma *(in* M_trancl*)* trancl_closed [intro,simp]:
"M(r) ==> M(trancl(r))"
⟨proof⟩

lemma *(in* M_trancl*)* trancl_abs [simp]:
"[| M(r); M(z) |] ==> tran_closure(M,r,z) <-> z = trancl(r)"
⟨proof⟩

lemma *(in* M_trancl*)* wellfounded_trancl_separation':
"[| M(r); M(Z) |] ==> separation (M, λx. ∃w[M]. w ∈ Z & <w,x> ∈ r^+)"
⟨proof⟩

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^+)"
⟨proof⟩

lemma *(in* M_trancl*)* wellfounded_on_trancl:
"[| wellfounded_on(M,A,r); r-``A <= A; M(r); M(A) |]
 ==> wellfounded_on(M,A,r^+)"
⟨proof⟩

lemma *(in* M_trancl*)* wellfounded_trancl:
"[|wellfounded(M,r); M(r)|] ==> wellfounded(M,r^+)"
⟨proof⟩

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})))"
```
⟨proof⟩

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))"
```

⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)>)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)>)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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}))>)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *bnd_mono_increasing [rule_format]:*
"[|i ∈ nat; j ∈ nat; bnd_mono(D,h)|] ==> i ≤ j --> h^i(0) ⊆ h^j(0)"
⟨*proof*⟩

lemma *directed_iterates: "bnd_mono(D,h) ==> directed({h^n (0). n∈nat})"*
⟨*proof*⟩

lemma *contin_iterates_eq:*
"[|bnd_mono(D, h); contin(h)|]
==> h(⋃n∈nat. h^n (0)) = (⋃n∈nat. h^n (0))"
⟨*proof*⟩

lemma *lfp_subset_Union:*
"[|bnd_mono(D, h); contin(h)|] ==> lfp(D,h) <= (⋃n∈nat. h^n(0))"
⟨*proof*⟩

lemma *Union_subset_lfp:*
"bnd_mono(D,h) ==> (⋃n∈nat. h^n(0)) <= lfp(D,h)"
⟨*proof*⟩

lemma *lfp_eq_Union:*
"[|bnd_mono(D, h); contin(h)|] ==> lfp(D,h) = (⋃n∈nat. h^n(0))"
⟨*proof*⟩

6.1.1 Some Standard Datatype Constructions Preserve Continuity

lemma *contin_imp_mono: "[|X⊆Y; contin(F)|] ==> F(X) ⊆ F(Y)"*
⟨*proof*⟩

lemma *sum_contin: "[|contin(F); contin(G)|] ==> contin(λX. F(X) + G(X))"*
⟨*proof*⟩

lemma *prod_contin: "[|contin(F); contin(G)|] ==> contin(λX. F(X) * G(X))"*

⟨*proof*⟩

lemma *const_contin: "contin(λX. A)"*
⟨*proof*⟩

lemma *id_contin: "contin(λX. X)"*
⟨*proof*⟩

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)"
```

⟨*proof*⟩

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)))"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *list_fun_contin: "contin(λX. {0} + A*X)"*
⟨*proof*⟩

Re-expresses lists using sum and product

lemma *list_eq_lfp2: "list(A) = lfp(univ(A), λX. {0} + A*X)"*
⟨*proof*⟩

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

lemma *list_eq_Union:*
*"list(A) = (⋃n∈nat. (λX. {0} + A*X) ^ n (0))"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

6.4 formulas without univ

lemma *formula_fun_bnd_mono:*
*"bnd_mono(univ(0), λX. ((nat*nat) + (nat*nat)) + (X*X + X))"*
⟨*proof*⟩

lemma *formula_fun_contin:*
*"contin(λX. ((nat*nat) + (nat*nat)) + (X*X + X))"*
⟨*proof*⟩

Re-expresses formulas using sum and product

lemma *formula_eq_lfp2:*
*"formula = lfp(univ(0), λX. ((nat*nat) + (nat*nat)) + (X*X + X))"*
⟨*proof*⟩

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

lemma *formula_eq_Union:*
"formula =
*(⋃n∈nat. (λX. ((nat*nat) + (nat*nat)) + (X*X + X)) ^ n (0))"*
⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *Cons_in_list_N [simp]:*

```
     "Cons(a,l) ∈ list_N(A,succ(n)) <-> a∈A & l ∈ list_N(A,n)"
```

⟨*proof*⟩

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

lemma *list_N_0: "list_N(A,0) = 0"*
⟨*proof*⟩

lemma *list_N_succ: "list_N(A,succ(n)) = {0} + A * (list_N(A,n))"*
⟨*proof*⟩

lemma *list_N_imp_list:*

```
  "[| l ∈ list_N(A,n); n ∈ nat |] ==> l ∈ list(A)"
```

⟨*proof*⟩

lemma *list_N_imp_length_lt [rule_format]:*

```
     "n ∈ nat ==> ∀l ∈ list_N(A,n). length(l) < n"
```

⟨*proof*⟩

lemma *list_imp_list_N [rule_format]:*

```
     "l ∈ list(A) ==> ∀n∈nat. length(l) < n --> l ∈ list_N(A, n)"
```

⟨*proof*⟩

lemma *list_N_imp_eq_length:*

```
      "[|n ∈ nat; l ∉ list_N(A, n); l ∈ list_N(A, succ(n))|]
       ==> n = length(l)"
```

⟨*proof*⟩

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"
```
⟨*proof*⟩

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"*
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *Equal_in_formula_N [simp]:*
```
"Equal(x,y) ∈ formula_N(succ(n)) <-> x ∈ nat & y ∈ nat"
```

⟨proof⟩

lemma *Nand_in_formula_N [simp]:*
```
     "Nand(x,y) ∈ formula_N(succ(n)) <-> x ∈ formula_N(n) & y ∈ formula_N(n)"
```
⟨proof⟩

lemma *Forall_in_formula_N [simp]:*
```
     "Forall(x) ∈ formula_N(succ(n)) <-> x ∈ formula_N(n)"
```
⟨proof⟩

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

lemma *formula_N_0: "formula_N(0) = 0"*
⟨proof⟩

lemma *formula_N_succ:*
```
     "formula_N(succ(n)) =
      ((nat*nat) + (nat*nat)) + (formula_N(n) * formula_N(n) + formula_N(n))"
```
⟨proof⟩

lemma *formula_N_imp_formula:*
```
  "[| p ∈ formula_N(n); n ∈ nat |] ==> p ∈ formula"
```
⟨proof⟩

lemma *formula_N_imp_depth_lt [rule_format]:*
```
     "n ∈ nat ==> ∀p ∈ formula_N(n). depth(p) < n"
```
⟨proof⟩

lemma *formula_imp_formula_N [rule_format]:*
```
     "p ∈ formula ==> ∀n∈nat. depth(p) < n --> p ∈ formula_N(n)"
```
⟨proof⟩

lemma *formula_N_imp_eq_depth:*
```
      "[|n ∈ nat; p ∉ formula_N(n); p ∈ formula_N(succ(n))|]
       ==> n = depth(p)"
```
⟨proof⟩

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)"
```
⟨proof⟩

lemma *formula_N_distrib:*
```
  "[| m ∈ nat; n ∈ nat |] ==> formula_N(m ∪ n) = formula_N(m) ∪ formula_N(n)"
```
⟨proof⟩

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))"
⟨proof⟩
```

```
lemma (in M_datatypes) list_closed [intro,simp]:
     "M(A) ==> M(list(A))"
⟨proof⟩
```

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)"
⟨proof⟩
```

```
lemma (in M_datatypes) list_N_closed [intro,simp]:
     "[|M(A); n∈nat|] ==> M(list_N(A,n))"
```

⟨*proof*⟩

lemma *(***in** *M_datatypes) mem_list_abs [simp]:*
    "M(A) ==> mem_list(M,A,l) <-> l ∈ list(A)"
⟨*proof*⟩

lemma *(***in** *M_datatypes) list_abs [simp]:*
    "[|M(A); M(Z)|] ==> is_list(M,A,Z) <-> Z = list(A)"
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *(***in** *M_datatypes) formula_closed [intro,simp]:*
    "M(formula)"
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *(***in** *M_datatypes) formula_N_closed [intro,simp]:*
    "n∈nat ==> M(formula_N(n))"
⟨*proof*⟩

lemma *(***in** *M_datatypes) mem_formula_abs [simp]:*
    "mem_formula(M,l) <-> l ∈ formula"
⟨*proof*⟩

lemma *(***in** *M_datatypes) formula_abs [simp]:*
    "[|M(Z)|] ==> is_formula(M,Z) <-> Z = formula"
⟨*proof*⟩

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))"
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *(***in** *M_eclose) eclose_closed [intro,simp]:*
```
     "M(A) ==> M(eclose(A))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *(***in** *M_eclose) mem_eclose_abs [simp]:*
```
     "M(A) ==> mem_eclose(M,A,l) <-> l ∈ eclose(A)"
```
⟨*proof*⟩

lemma *(***in** *M_eclose) eclose_abs [simp]:*
```
     "[|M(A); M(Z)|] ==> is_eclose(M,A,Z) <-> Z = eclose(A)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

Proof is trivial since `length` returns natural numbers.

lemma *(***in** *M_trivial) length_closed [intro,simp]:*
```
"l ∈ list(A) ==> M(length(l))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *(***in** *M_basic) iterates_tl'_closed:*
```
"[|n ∈ nat; M(x)|] ==> M(tl'^n (x))"
```
⟨*proof*⟩

Immediate by type-checking

lemma *(***in** *M_datatypes) nth_closed [intro,simp]:*
```
"[|xs ∈ list(A); n ∈ nat; M(A)|] ==> M(nth(n,xs))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *(***in** *M_trivial) Member_in_M_iff [iff]:*
```
"M(Member(x,y)) <-> M(x) & M(y)"
```
⟨*proof*⟩

definition
is_Equal :: "[i=>o,i,i,i] => o" **where**
— because *Equal(x, y)* $\equiv$ *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))"
⟨*proof*⟩

lemma (**in** *M_trivial*) *Equal_in_M_iff [iff]*: "M(Equal(x,y)) <-> M(x) & M(y)"
⟨*proof*⟩

definition
is_Nand :: "[i=>o,i,i,i] => o" **where**
— because *Nand(x, y)* $\equiv$ *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))"
⟨*proof*⟩

lemma (**in** *M_trivial*) *Nand_in_M_iff [iff]*: "M(Nand(x,y)) <-> M(x) & M(y)"
⟨*proof*⟩

definition
is_Forall :: "[i=>o,i,i] => o" **where**
— because *Forall(x)* $\equiv$ *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))"
⟨*proof*⟩

lemma (**in** *M_trivial*) *Forall_in_M_iff [iff]*: "M(Forall(x)) <-> M(x)"
⟨*proof*⟩

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"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

Proof is trivial since `depth` returns natural numbers.

lemma *(in* M_trivial*)* depth_closed [intro,simp]:
```
     "p ∈ formula ==> M(depth(p))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))"
⟨proof⟩
```

```
lemma (in Formula_Rec) formula_rec_lam_closed:
    "M(g) ==> M(Lambda (formula, formula_rec_case(a,b,c,d,g)))"
⟨proof⟩
```

```
lemma (in Formula_Rec) MH_rel2:
     "relation2 (M, MH,
             λx h. Lambda (formula, formula_rec_case(a,b,c,d,h)))"
⟨proof⟩
```

```
lemma (in Formula_Rec) fr_transrec_closed:
    "n ∈ nat
     ==> M(transrec
          (n, λx h. Lambda(formula, formula_rec_case(a, b, c, d, h))))"
⟨proof⟩
```

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))"
⟨proof⟩
```

```
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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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

      ==> P(⋃(I))"
```
⟨*proof*⟩

lemma *UnboundedD:*
```
     "[| Unbounded(P);  Ord(i) |] ==> ∃j. i<j ∧ P(j)"
```
⟨*proof*⟩

lemma *Closed_Unbounded_imp_Unbounded:* `"Closed_Unbounded(C) ==> Unbounded(C)"`

⟨*proof*⟩

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)"
```
⟨*proof*⟩

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

```
theorem Closed_Unbounded_Ord   [simp]: "Closed_Unbounded(Ord)"
```
⟨*proof*⟩

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

```
theorem Closed_Unbounded_Limit [simp]: "Closed_Unbounded(Limit)"
```
⟨*proof*⟩

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

```
theorem Closed_Unbounded_Card  [simp]: "Closed_Unbounded(Card)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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

```
lemma (in cub_family) Ord_sup_greater:
     "Ord(sup_greater(x))"
```
⟨*proof*⟩

```
lemma (in cub_family) Ord_next_greater:
     "Ord(next_greater(a,x))"
```
⟨*proof*⟩

`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)"
⟨proof⟩

lemma (**in** cub_family) next_greater_in_P:
"[| Ord(x); a∈A |] ==> P(a, next_greater(a,x))"
⟨proof⟩

lemma (**in** cub_family) next_greater_gt:
"[| Ord(x); a∈A |] ==> x < next_greater(a,x)"
⟨proof⟩

lemma (**in** cub_family) sup_greater_gt:
"Ord(x) ==> x < sup_greater(x)"
⟨proof⟩

lemma (**in** cub_family) next_greater_le_sup_greater:
"a∈A ==> next_greater(a,x) ≤ sup_greater(x)"
⟨proof⟩

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)))"
⟨proof⟩

lemma (**in** cub_family) P_omega_sup_greater:
"[| Ord(x); a∈A |] ==> P(a, sup_greater^ω (x))"
⟨proof⟩

lemma (**in** cub_family) omega_sup_greater_gt:
"Ord(x) ==> x < sup_greater^ω (x)"
⟨proof⟩

lemma (**in** cub_family) Unbounded_INT: "Unbounded(λx. ∀a∈A. P(a,x))"
⟨proof⟩

lemma (**in** cub_family) Closed_Unbounded_INT:
"Closed_Unbounded(λx. ∀a∈A. P(a,x))"
⟨proof⟩

theorem Closed_Unbounded_INT:
"(!!a. a∈A ==> Closed_Unbounded(P(a)))
 ==> Closed_Unbounded(λx. ∀a∈A. P(a, x))"
⟨proof⟩

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

⟨proof⟩

theorem *Closed_Unbounded_Int:*
```
    "[| Closed_Unbounded(P); Closed_Unbounded(Q) |]
     ==> Closed_Unbounded(λx. P(x) ∧ Q(x))"
```
⟨proof⟩

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)"
```
⟨proof⟩

lemma *mono_Ord_imp_Ord:* `"[| Ord(i); mono_Ord(F) |] ==> Ord(F(i))"`
⟨proof⟩

lemma *mono_Ord_imp_mono:* `"[| i<j; mono_Ord(F) |] ==> F(i) < F(j)"`
⟨proof⟩

lemma *Normal_imp_Ord [simp]:* `"[| Normal(F); Ord(i) |] ==> Ord(F(i))"`
⟨proof⟩

lemma *Normal_imp_cont:* `"[| Normal(F); Limit(l) |] ==> F(l) = (⋃i<l. F(i))"`
⟨proof⟩

lemma *Normal_imp_mono:* `"[| i<j; Normal(F) |] ==> F(i) < F(j)"`
⟨proof⟩

lemma *Normal_increasing:* `"[| Ord(i); Normal(F) |] ==> i≤F(i)"`

⟨*proof*⟩

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)"*
⟨*proof*⟩

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; $\forall x \in X$. Ord(x) |]
==> F(Union(X)) = ($\bigcup y \in X$. F(y))"
⟨*proof*⟩

lemma *Normal_Union:*
"[| X$\neq$0; $\forall x \in X$. Ord(x); Normal(F) |] ==> F(Union(X)) = ($\bigcup y \in X$. F(y))"
⟨*proof*⟩

lemma *Normal_imp_fp_Closed: "Normal(F) ==> Closed(λi. F(i) = i)"*
⟨*proof*⟩

lemma *iterates_Normal_increasing:*
"[| n$\in$nat; x < F(x); Normal(F) |]
==> F^n (x) < F^(succ(n)) (x)"
⟨*proof*⟩

lemma *Ord_iterates_Normal:*
"[| n$\in$nat; Normal(F); Ord(x) |] ==> Ord(F^n (x))"
⟨*proof*⟩

THIS RESULT IS UNUSED

lemma *iterates_omega_Limit:*
"[| Normal(F); x < F(x) |] ==> Limit(F^ω (x))"
⟨*proof*⟩

lemma *iterates_omega_fixedpoint:*
"[| Normal(F); Ord(a) |] ==> F(F^ω (a)) = F^ω (a)"
⟨*proof*⟩

lemma *iterates_omega_increasing:*
"[| Normal(F); Ord(a) |] ==> a $\leq$ F^ω (a)"
⟨*proof*⟩

lemma *Normal_imp_fp_Unbounded: "Normal(F) ==> Unbounded(λi. F(i) = i)"*
⟨*proof*⟩

theorem *Normal_imp_fp_Closed_Unbounded:*
```
     "Normal(F) ==> Closed_Unbounded(λi. F(i) = i)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *normalize_lemma [rule_format]:*
```
     "[| Ord(b); !!x. Ord(x) ==> Ord(F(x)) |]
      ==> ∀a. a < b --> normalize(F, a) < normalize(F, b)"
```
⟨*proof*⟩

lemma *normalize_increasing:*
```
     "[| a < b;  !!x. Ord(x) ==> Ord(F(x)) |]
      ==> normalize(F, a) < normalize(F, b)"
```
⟨*proof*⟩

theorem *Normal_normalize:*
```
     "(!!x. Ord(x) ==> Ord(F(x))) ==> Normal(normalize(F))"
```
⟨*proof*⟩

theorem *le_normalize:*
```
     "[| Ord(a); cont_Ord(F); !!x. Ord(x) ==> Ord(F(x)) |]
      ==> F(a) ≤ normalize(F,a)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *Aleph_lemma [rule_format]:*
```
     "Ord(b) ==> ∀a. a < b --> Aleph(a) < Aleph(b)"
```
⟨*proof*⟩

lemma *Aleph_increasing:*
```
     "a < b ==> Aleph(a) < Aleph(b)"
```
⟨*proof*⟩

theorem *Normal_Aleph: "Normal(Aleph)"*
⟨*proof*⟩

end

8 The Reflection Theorem

theory *Reflection* **imports** *Normal* **begin**

lemma *all_iff_not_ex_not: "(∀x. P(x)) <-> (~ (∃x. ~ P(x)))"*
⟨*proof*⟩

lemma *ball_iff_not_bex_not: "(∀x∈A. P(x)) <-> (~ (∃x∈A. ~ P(x)))"*
⟨*proof*⟩

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)"
⟨proof⟩
```

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"
⟨proof⟩
```

8.2 Easy Cases of the Reflection Theorem

```
theorem (in reflection) Triv_reflection [intro]:
     "Reflects(Ord, P, λa x. P(x))"
⟨proof⟩

theorem (in reflection) Not_reflection [intro]:
     "Reflects(Cl,P,Q) ==> Reflects(Cl, λx. ~P(x), λa x. ~Q(a,x))"
⟨proof⟩

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))"
⟨proof⟩

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))"
⟨proof⟩

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))"
⟨proof⟩

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))"
```
⟨proof⟩

8.3 Reflection for Existential Quantifiers

lemma *(in reflection) FO_works:*
```
     "[| y∈Mset(a); Ord(a); M(z); P(<y,z>) |] ==> ∃z∈Mset(FO(P,y)).
P(<y,z>)"
```
⟨proof⟩

lemma *(in reflection) Ord_FO [intro,simp]: "Ord(FO(P,y))"*
⟨proof⟩

lemma *(in reflection) Ord_FF [intro,simp]: "Ord(FF(P,y))"*
⟨proof⟩

lemma *(in reflection) cont_Ord_FF: "cont_Ord(FF(P))"*
⟨proof⟩

Recall that *FO* depends upon $y \in$ *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>)"
```
⟨proof⟩

lemma *(in reflection) FFN_works:*
```
     "[| M(z); y∈Mset(a); P(<y,z>); Ord(a) |]
      ==> ∃z∈Mset(normalize(FF(P),a)). P(<y,z>)"
```
⟨proof⟩

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>)"
```
⟨proof⟩

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>)"
```

⟨*proof*⟩

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>))"
```
⟨*proof*⟩

...and it is closed and unbounded

lemma *(in* `ex_reflection`*)* `ZF_Closed_Unbounded_ClEx:`
```
    "Closed_Unbounded(ClEx(P))"
```
⟨*proof*⟩

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>))"
```
⟨*proof*⟩

lemma *(in* `reflection`*)* `Closed_Unbounded_ClEx:`
```
    "(!!a. [| Cl(a); Ord(a) |] ==> ∀x∈Mset(a). P(x) <-> Q(a,x))
     ==> Closed_Unbounded(ClEx(P))"
```
⟨*proof*⟩

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>))"
```
⟨*proof*⟩

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>))"
```
⟨*proof*⟩

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))"
```
⟨proof⟩

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))"
```
⟨proof⟩

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))"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

Example 3'

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

Example 3"

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

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)))"
```
⟨*proof*⟩

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*⟩

lemma *meta_exE: "[| ?? x. PROP P(x); !!x. PROP P(x) ==> PROP R |] ==> PROP R"*
⟨*proof*⟩

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)"*

⟨proof⟩

lemma *nonempty: "L(0)"*
⟨proof⟩

theorem *upair_ax: "upair_ax(L)"*
⟨proof⟩

theorem *Union_ax: "Union_ax(L)"*
⟨proof⟩

theorem *power_ax: "power_ax(L)"*
⟨proof⟩

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

theorem *foundation_ax: "foundation_ax(L)"*
⟨proof⟩

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)"
```
⟨proof⟩

lemma *LReplace_in_L:*
```
    "[|L(X); univalent(L,X,Q)|]
     ==> ∃Y. L(Y) & Replace(X, %x y. Q(x,y) & L(y)) ⊆ Y"
```
⟨proof⟩

theorem *replacement: "replacement(L,P)"*
⟨proof⟩

10.2 Instantiating the locale *M_trivial*

No instances of Separation yet.

lemma *Lset_mono_le: "mono_le_subset(Lset)"*
⟨proof⟩

lemma *Lset_cont: "cont_Ord(Lset)"*
⟨proof⟩

lemmas *L_nat = Ord_in_L [OF Ord_nat]*

theorem *M_trivial_L: "PROP M_trivial(L)"*
⟨proof⟩

interpretation *M_trivial ["L"]* ⟨proof⟩

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)]"
```
⟨*proof*⟩

theorem *Not_reflection:*
```
     "REFLECTS[P,Q] ==> REFLECTS[λx. ~P(x), λa x. ~Q(a,x)]"
```
⟨*proof*⟩

theorem *And_reflection:*
```
     "[| REFLECTS[P,Q]; REFLECTS[P',Q'] |]
      ==> REFLECTS[λx. P(x) ∧ P'(x), λa x. Q(a,x) ∧ Q'(a,x)]"
```
⟨*proof*⟩

theorem *Or_reflection:*
```
     "[| REFLECTS[P,Q]; REFLECTS[P',Q'] |]
      ==> REFLECTS[λx. P(x) ∨ P'(x), λa x. Q(a,x) ∨ Q'(a,x)]"
```
⟨*proof*⟩

theorem *Imp_reflection:*
```
     "[| REFLECTS[P,Q]; REFLECTS[P',Q'] |]
      ==> REFLECTS[λx. P(x) --> P'(x), λa x. Q(a,x) --> Q'(a,x)]"
```
⟨*proof*⟩

theorem *Iff_reflection:*
```
     "[| REFLECTS[P,Q]; REFLECTS[P',Q'] |]
      ==> REFLECTS[λx. P(x) <-> P'(x), λa x. Q(a,x) <-> Q'(a,x)]"
```

⟨proof⟩

lemma *reflection_Lset: "reflection(Lset)"*
⟨proof⟩

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)]"
```
⟨proof⟩

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)]"
```
⟨proof⟩

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)]"
```
⟨proof⟩

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)]"
```
⟨proof⟩

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)]"
```
⟨proof⟩

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)]"
```
⟨proof⟩

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))"
```
⟨proof⟩

lemma *ReflectsE:*
```
"[| REFLECTS[P,Q]; Ord(i);
    !!j. [|i<j;  ∀x ∈ Lset(j). P(x) <-> Q(j,x)|] ==> R |]
 ==> R"
```
⟨*proof*⟩

lemma *Collect_mem_eq: "{x∈A. x∈B} = A ∩ B"*
⟨*proof*⟩

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"
⟨*proof*⟩

lemma *sats_empty_fm [simp]:*
```
"[| x ∈ nat; env ∈ list(A)|]
 ==> sats(A, empty_fm(x), env) <-> empty(##A, nth(x,env))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *empty_reflection:*
```
     "REFLECTS[λx. empty(L,f(x)),
               λi x. empty(##Lset(i),f(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)}"
```
⟨*proof*⟩

theorem *upair_reflection:*
```
     "REFLECTS[λx. upair(L,f(x),g(x),h(x)),
               λi x. upair(##Lset(i),f(x),g(x),h(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *pair_reflection:*
```
     "REFLECTS[λx. pair(L,f(x),g(x),h(x)),
               λi x. pair(##Lset(i),f(x),g(x),h(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

theorem *union_reflection:*
```
     "REFLECTS[λx. union(L,f(x),g(x),h(x)),
               λi x. union(##Lset(i),f(x),g(x),h(x))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *successor_reflection:*
```
    "REFLECTS[λx. successor(L,f(x),g(x)),
              λi x. successor(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *sats_number1_fm [simp]:*
```
  "[| x ∈ nat; env ∈ list(A)|]
   ==> sats(A, number1_fm(x), env) <-> number1(##A, nth(x,env))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *number1_reflection:*
```
    "REFLECTS[λx. number1(L,f(x)),
              λi x. number1(##Lset(i),f(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *big_union_reflection:*
```
"REFLECTS[λx. big_union(L,f(x),g(x)),
          λi x. big_union(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

theorem *subset_reflection:*
```
"REFLECTS[λx. subset(L,f(x),g(x)),
          λi x. subset(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

lemma *sats_transset_fm':*
```
"[|x ∈ nat; env ∈ list(A)|]
 ==> sats(A, transset_fm(x), env) <-> transitive_set(##A, nth(x,env))"
```

⟨*proof*⟩

theorem *transitive_set_reflection:*
```
     "REFLECTS[λx. transitive_set(L,f(x)),
               λi x. transitive_set(##Lset(i),f(x))]"
```
⟨*proof*⟩

lemma *sats_ordinal_fm':*
```
   "[|x ∈ nat; env ∈ list(A)|]
    ==> sats(A, ordinal_fm(x), env) <-> ordinal(##A,nth(x,env))"
```
⟨*proof*⟩

lemma *ordinal_iff_sats:*
```
      "[| nth(i,env) = x;  i ∈ nat; env ∈ list(A)|]
       ==> ordinal(##A, x) <-> sats(A, ordinal_fm(i), env)"
```
⟨*proof*⟩

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

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *membership_reflection:*

```
"REFLECTS[λx. membership(L,f(x),g(x)),
          λi x. membership(##Lset(i),f(x),g(x))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *domain_reflection:*
```
"REFLECTS[λx. is_domain(L,f(x),g(x)),
          λi x. is_domain(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *range_reflection:*
```
"REFLECTS[λx. is_range(L,f(x),g(x)),
          λi x. is_range(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

theorem *field_reflection:*

```
     "REFLECTS[λx. is_field(L,f(x),g(x)),
               λi x. is_field(##Lset(i),f(x),g(x))]"
```

⟨*proof*⟩

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"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *image_reflection:*
```
     "REFLECTS[λx. image(L,f(x),g(x),h(x)),
               λi x. image(##Lset(i),f(x),g(x),h(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *sats_relation_fm [simp]:*
```
  "[| x ∈ nat; env ∈ list(A)|]
   ==> sats(A, relation_fm(x), env) <-> is_relation(##A, nth(x,env))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *is_relation_reflection:*
```
    "REFLECTS[λx. is_relation(L,f(x)),
              λi x. is_relation(##Lset(i),f(x))]"
```
⟨*proof*⟩

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"
```

⟨*proof*⟩

lemma *sats_function_fm [simp]:*

```
   "[| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, function_fm(x), env) <-> is_function(##A, nth(x,env))"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

theorem *is_function_reflection:*

```
     "REFLECTS[λx. is_function(L,f(x)),
               λi x. is_function(##Lset(i),f(x))]"
```

⟨*proof*⟩

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"
```

⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```

⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *composition_reflection:*
```
"REFLECTS[λx. composition(L,f(x),g(x),h(x)),
          λi x. composition(##Lset(i),f(x),g(x),h(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *injection_reflection:*
```
    "REFLECTS[λx. injection(L,f(x),g(x),h(x)),
              λi x. injection(##Lset(i),f(x),g(x),h(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *surjection_reflection:*
```
    "REFLECTS[λx. surjection(L,f(x),g(x),h(x)),
              λi x. surjection(##Lset(i),f(x),g(x),h(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```

⟨proof⟩

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)"
```
⟨proof⟩

theorem *bijection_reflection:*
```
     "REFLECTS[λx. bijection(L,f(x),g(x),h(x)),
               λi x. bijection(##Lset(i),f(x),g(x),h(x))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

theorem *restriction_reflection:*
```
     "REFLECTS[λx. restriction(L,f(x),g(x),h(x)),
               λi x. restriction(##Lset(i),f(x),g(x),h(x))]"
```
⟨proof⟩

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"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))]"
```

⟨*proof*⟩

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"
```

⟨proof⟩

lemma *sats_limit_ordinal_fm [simp]:*

```
"[| x ∈ nat; env ∈ list(A)|]
 ==> sats(A, limit_ordinal_fm(x), env) <-> limit_ordinal(##A, nth(x,env))"
```

⟨proof⟩

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)"
```

⟨proof⟩

theorem *limit_ordinal_reflection:*

```
"REFLECTS[λx. limit_ordinal(L,f(x)),
          λi x. limit_ordinal(##Lset(i),f(x))]"
```

⟨proof⟩

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"
```

⟨proof⟩

lemma *sats_finite_ordinal_fm [simp]:*

```
"[| x ∈ nat; env ∈ list(A)|]
 ==> sats(A, finite_ordinal_fm(x), env) <-> finite_ordinal(##A, nth(x,env))"
```

⟨proof⟩

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)"
```

⟨proof⟩

theorem *finite_ordinal_reflection:*

```
"REFLECTS[λx. finite_ordinal(L,f(x)),
          λi x. finite_ordinal(##Lset(i),f(x))]"
```

⟨proof⟩

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"
```
⟨*proof*⟩

lemma *sats_omega_fm [simp]:*
```
   "[| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, omega_fm(x), env) <-> omega(##A, nth(x,env))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *omega_reflection:*
```
     "REFLECTS[λx. omega(L,f(x)),
               λi x. omega(##Lset(i),f(x))]"
```
⟨*proof*⟩

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"
⟨proof⟩

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)"
⟨proof⟩

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))"
⟨proof⟩

lemma separation_CollectI:
     "(⋀z. L(z) ==> L({x ∈ z . P(x)})) ==> separation(L, λx. P(x))"
⟨proof⟩
```

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)})"
⟨proof⟩
```

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)"
⟨proof⟩
```

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)"
⟨proof⟩
```

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]"
⟨proof⟩

lemma Inter_separation:
     "L(A) ==> separation(L, λx. ∀y[L]. y∈A --> x∈y)"
⟨proof⟩
```

11.2 Separation for Set Difference

```
lemma Diff_Reflects:
     "REFLECTS[λx. x ∉ B, λi x. x ∉ B]"
⟨proof⟩

lemma Diff_separation:
     "L(B) ==> separation(L, λx. x ∉ B)"
⟨proof⟩
```

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))]"
⟨proof⟩

lemma cartprod_separation:
     "[| L(A); L(B) |]
      ==> separation(L, λz. ∃x[L]. x∈A & (∃y[L]. y∈B & pair(L,x,y,z)))"
⟨proof⟩
```

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))]"
⟨proof⟩

lemma image_separation:
```

```
    "[| L(A); L(r) |]
     ==> separation(L, λy. ∃p[L]. p∈r & (∃x[L]. x∈A & pair(L,x,y,p)))"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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)))"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

lemma *restrict_separation:*
```
  "L(A) ==> separation(L, λz. ∃x[L]. x∈A & (∃y[L]. pair(L,x,y,z)))"
```
⟨*proof*⟩

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]"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)]"
```
⟨*proof*⟩

lemma *pred_separation:*
```
     "[| L(r); L(x) |] ==> separation(L, λy. ∃p[L]. p∈r & pair(L,y,x,p))"
```
⟨*proof*⟩

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]"
```
⟨*proof*⟩

lemma *Memrel_separation:*
```
     "separation(L, λz. ∃x[L]. ∃y[L]. pair(L,x,y,z) & x ∈ y)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)]"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"*
⟨*proof*⟩

theorem *M_basic_L: "PROP M_basic(L)"*
⟨*proof*⟩

interpretation *M_basic [L]* ⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

theorem *Inl_reflection:*
```
     "REFLECTS[λx. is_Inl(L,f(x),h(x)),
               λi x. is_Inl(##Lset(i),f(x),h(x))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

theorem *Inr_reflection:*
```
     "REFLECTS[λx. is_Inr(L,f(x),h(x)),
               λi x. is_Inr(##Lset(i),f(x),h(x))]"
```
⟨proof⟩

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"*
⟨proof⟩

lemma *sats_Nil_fm [simp]:*

```
   "[| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, Nil_fm(x), env) <-> is_Nil(##A, nth(x,env))"
```
⟨proof⟩

lemma *Nil_iff_sats:*
```
      "[| nth(i,env) = x; i ∈ nat; env ∈ list(A)|]
       ==> is_Nil(##A, x) <-> sats(A, Nil_fm(i), env)"
```
⟨proof⟩

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

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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"`

⟨proof⟩

lemma *sats_quasilist_fm [simp]:*
```
   "[| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, quasilist_fm(x), env) <-> is_quasilist(##A, nth(x,env))"
```
⟨proof⟩

lemma *quasilist_iff_sats:*
```
      "[| nth(i,env) = x; i ∈ nat; env ∈ list(A)|]
       ==> is_quasilist(##A, x) <-> sats(A, quasilist_fm(i), env)"
```
⟨proof⟩

theorem *quasilist_reflection:*
```
     "REFLECTS[λx. is_quasilist(L,f(x)),
               λi x. is_quasilist(##Lset(i),f(x))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

theorem *hd_reflection:*
```
     "REFLECTS[λx. is_hd(L,f(x),g(x)),
               λi x. is_hd(##Lset(i),f(x),g(x))]"
```
⟨proof⟩

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"
⟨proof⟩

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))"
⟨proof⟩

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)"
⟨proof⟩

theorem tl_reflection:
     "REFLECTS[λx. is_tl(L,f(x),g(x)),
               λi x. is_tl(##Lset(i),f(x),g(x))]"
⟨proof⟩
```

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"
⟨proof⟩

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))"
⟨proof⟩
```

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))]"
```

⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *Forall_reflection:*
```
     "REFLECTS[λx. is_Forall(L,f(x),g(x)),
               λi x. is_Forall(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
⟨*proof*⟩

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))"
⟨*proof*⟩

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)"
⟨*proof*⟩

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))]"
⟨*proof*⟩

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"
⟨*proof*⟩

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))"
⟨*proof*⟩

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)"
⟨*proof*⟩

theorem *is_not_reflection:*

```
    "REFLECTS[λx. is_not(L,f(x),g(x)),
              λi x. is_not(##Lset(i),f(x),g(x))]"
```
⟨proof⟩

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)))"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *cartprod_reflection:*
```
"REFLECTS[λx. cartprod(L,f(x),g(x),h(x)),
          λi x. cartprod(##Lset(i),f(x),g(x),h(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *sats_quasinat_fm [simp]:*
```
   "[| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, quasinat_fm(x), env) <-> is_quasinat(##A, nth(x,env))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *quasinat_reflection:*
```
     "REFLECTS[λx. is_quasinat(L,f(x)),
               λi x. is_quasinat(##Lset(i),f(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
⟨proof⟩
```

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"
⟨proof⟩

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))"
⟨proof⟩

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)"
⟨proof⟩

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))]"
⟨proof⟩
```

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"
⟨proof⟩

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))"
⟨proof⟩
```

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)"
```
⟨*proof*⟩

theorem *mem_eclose_reflection:*
```
     "REFLECTS[λx. mem_eclose(L,f(x),g(x)),
               λi x. mem_eclose(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *is_eclose_reflection:*
```
     "REFLECTS[λx. is_eclose(L,f(x),g(x)),
               λi x. is_eclose(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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"
⟨*proof*⟩

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))"
⟨*proof*⟩

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)"
⟨*proof*⟩

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))]"
⟨*proof*⟩

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"
⟨*proof*⟩

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))"
⟨*proof*⟩

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)"
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *mem_list_reflection:*
```
     "REFLECTS[λx. mem_list(L,f(x),g(x)),
              λi x. mem_list(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨proof⟩

theorem *is_list_reflection:*
```
    "REFLECTS[λx. is_list(L,f(x),g(x)),
             λi x. is_list(##Lset(i),f(x),g(x))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *sats_mem_formula_fm [simp]:*
```
   "[| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, mem_formula_fm(x), env) <-> mem_formula(##A, nth(x,env))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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

⟨proof⟩

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"
```
⟨proof⟩

lemma *sats_is_formula_fm [simp]:*
```
   "[| x ∈ nat; env ∈ list(A)|]
    ==> sats(A, is_formula_fm(x), env) <-> is_formula(##A, nth(x,env))"
```
⟨proof⟩

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)"
```
⟨proof⟩

theorem *is_formula_reflection:*
```
     "REFLECTS[λx. is_formula(L,f(x)),
               λi x. is_formula(##Lset(i),f(x))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```

⟨proof⟩

```
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)"
```

⟨proof⟩

```
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))]"
```

⟨proof⟩

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"*
⟨proof⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

Separation for `r^*`.

lemma *rtrancl_separation:*
```
     "[| L(r); L(A) |] ==> separation (L, rtran_closure_mem(L,A,r))"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

theorem *rtran_closure_reflection:*
```
     "REFLECTS[λx. rtran_closure(L,f(x),g(x)),
               λi x. rtran_closure(##Lset(i),f(x),g(x))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨proof⟩

theorem *tran_closure_reflection:*

```
    "REFLECTS[λx. tran_closure(L,f(x),g(x)),
              λi x. tran_closure(##Lset(i),f(x),g(x))]"
```
⟨proof⟩

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]"
```
⟨proof⟩

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)"
```
⟨proof⟩

12.1.5 Instantiating the locale *M_trancl*

lemma *M_trancl_axioms_L: "M_trancl_axioms(L)"*
⟨proof⟩

theorem *M_trancl_L: "PROP M_trancl(L)"*
⟨proof⟩

interpretation *M_trancl [L]* ⟨proof⟩

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))]"
⟨proof⟩

lemma *list_replacement1:*
"L(A) ==> iterates_replacement(L, is_list_functor(L,A), 0)"
⟨proof⟩

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)]"
⟨proof⟩

lemma *list_replacement2:*
"L(A) ==> strong_replacement(L,
λn y. n∈nat & is_iterates(L, is_list_functor(L,A), 0, n, y))"
⟨proof⟩

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))]"
⟨proof⟩

lemma *formula_replacement1:*
"iterates_replacement(L, is_formula_functor(L), 0)"
⟨proof⟩

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)]"
```
⟨proof⟩

lemma *formula_replacement2:*
```
   "strong_replacement(L,
         λn y. n∈nat & is_iterates(L, is_formula_functor(L), 0, n, y))"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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))]"
```
⟨*proof*⟩

lemma *nth_replacement:*
```
   "L(w) ==> iterates_replacement(L, is_tl(L), w)"
```
⟨*proof*⟩

12.3.4 Instantiating the locale *M_datatypes*

lemma *M_datatypes_axioms_L: "M_datatypes_axioms(L)"*
⟨*proof*⟩

theorem *M_datatypes_L: "PROP M_datatypes(L)"*
⟨*proof*⟩

interpretation *M_datatypes [L]* ⟨*proof*⟩

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))]"
```
⟨*proof*⟩

lemma *eclose_replacement1:*
```
   "L(A) ==> iterates_replacement(L, big_union(L), A)"
```
⟨*proof*⟩

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)]"
```
⟨*proof*⟩

lemma *eclose_replacement2:*
```
   "L(A) ==> strong_replacement(L,
         λn y. n∈nat & is_iterates(L, big_union(L), A, n, y))"
```

⟨proof⟩

12.4.2 Instantiating the locale *M_eclose*

lemma *M_eclose_axioms_L: "M_eclose_axioms(L)"*
⟨proof⟩

theorem *M_eclose_L: "PROP M_eclose(L)"*
⟨proof⟩

interpretation *M_eclose [L]* ⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```

⟨proof⟩

theorem *depth_reflection:*
```
     "REFLECTS[λx. is_depth(L, f(x), g(x)),
               λi x. is_depth(##Lset(i), f(x), g(x))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
⟨proof⟩
```

```
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)"
⟨proof⟩
```

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))⟩)"
⟨proof⟩

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))⟩)"
```

⟨proof⟩

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)⟩)"
```
⟨proof⟩

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)⟩)"
```
⟨proof⟩

lemma *(in M_satisfies) a_closed:*
```
     "[|M(A); x∈nat; y∈nat|] ==> M(satisfies_a(A,x,y))"
```
⟨proof⟩

lemma *(in M_satisfies) a_rel:*
```
     "M(A) ==> Relation2(M, nat, nat, satisfies_is_a(M,A), satisfies_a(A))"
```
⟨proof⟩

lemma *(in M_satisfies) b_closed:*
```
     "[|M(A); x∈nat; y∈nat|] ==> M(satisfies_b(A,x,y))"
```
⟨proof⟩

lemma *(in M_satisfies) b_rel:*
```
     "M(A) ==> Relation2(M, nat, nat, satisfies_is_b(M,A), satisfies_b(A))"
```
⟨proof⟩

lemma *(in M_satisfies) c_closed:*
```
     "[|M(A); x ∈ formula; y ∈ formula; M(rx); M(ry)|]
      ==> M(satisfies_c(A,x,y,rx,ry))"
```
⟨proof⟩

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))"
```
⟨proof⟩

lemma *(in M_satisfies) d_closed:*
```
     "[|M(A); x ∈ formula; M(rx)|] ==> M(satisfies_d(A,x,rx))"
```
⟨proof⟩

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))"
```
⟨*proof*⟩

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

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))"
```
⟨*proof*⟩

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)⟩)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

```
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))"
⟨proof⟩

lemma (in M_satisfies) satisfies_abs:
  "[|M(A); M(z); p ∈ formula|]
   ==> is_satisfies(M,A,p,z) <-> z = satisfies(A,p)"
⟨proof⟩
```

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"
⟨proof⟩

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))"
⟨proof⟩

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)"
⟨proof⟩

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))]"
⟨proof⟩
```

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```

⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```

⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```

⟨*proof*⟩

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))]"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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))]"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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))]"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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))]"
```

⟨proof⟩

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))"
```

⟨proof⟩

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))]"
```

⟨proof⟩

lemma *formula_rec_replacement:*

— For the *transrec*

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

⟨proof⟩

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))]"
```

⟨*proof*⟩

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)))"
```
⟨*proof*⟩

13.5 Instantiating *M_satisfies*

lemma *M_satisfies_axioms_L: "M_satisfies_axioms(L)"*
⟨*proof*⟩

theorem *M_satisfies_L: "PROP M_satisfies(L)"*
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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))]"
```

⟨*proof*⟩

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))}}"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

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))}"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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)))"
```
⟨*proof*⟩

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))})"
```

⟨*proof*⟩

lemma *univalent_pair_eq:*
```
     "univalent (M, A, λxy z. ∃x∈B. ∃y∈C. xy = ⟨x,y⟩ ∧ z = f(x,y))"
```
⟨*proof*⟩

lemma *(***in*** M_DPow) DPow'_closed: "M(A) ==> M(DPow'(A))"*
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))]"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

14.4.3 Actually Instantiating the Locale

lemma *M_DPow_axioms_L: "M_DPow_axioms(L)"*
⟨*proof*⟩

theorem *M_DPow_L: "PROP M_DPow(L)"*
⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
⟨proof⟩

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)"
⟨proof⟩
```

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))]"
⟨proof⟩
```

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"
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

theorem *DPow'_reflection:*
```
    "REFLECTS[λx. is_DPow'(L,f(x),g(x)),
              λi x. is_DPow'(##Lset(i),f(x),g(x))]"
```
⟨proof⟩

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)"
```
⟨proof⟩

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))"
```
⟨proof⟩

lemma *(in M_Lset) DPow_apply_closed:*
```
   "[|M(f); M(x); y∈x|] ==> M(DPow'(f`y))"
```
⟨proof⟩

lemma *(in M_Lset) RepFun_DPow_apply_closed:*
```
   "[|M(f); M(x)|] ==> M({DPow'(f`y). y∈x})"
```
⟨proof⟩

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}"
```
⟨proof⟩

lemma *(in M_Lset) transrec_rep':*

```
   "M(i) ==> transrec_replacement(M, λx f u. u = (⋃y∈x. DPow'(f ` y)),
i)"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

lemma *(in M_Lset) Lset_closed:*

```
  "[|Ord(i);  M(i)|] ==> M(Lset(i))"
```

⟨*proof*⟩

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))]"
```

⟨*proof*⟩

lemma *strong_rep:*

```
   "[|L(x); L(g)|] ==> strong_replacement(L, λy z. transrec_body(L,g,x,y,z))"
```

⟨*proof*⟩

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))]"
```
⟨proof⟩

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)"
```
⟨proof⟩

14.6.3 Actually Instantiating *M_Lset*

lemma *M_Lset_axioms_L: "M_Lset_axioms(L)"*
⟨proof⟩

theorem *M_Lset_L: "PROP M_Lset(L)"*
⟨proof⟩

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)"
```
⟨proof⟩

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)"
```
⟨*proof*⟩

lemma `linear_rlist:`
```
    "linear(A,r) ==> linear(list(A),rlist(A,r))"
```
⟨*proof*⟩

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)"`
⟨*proof*⟩

lemma *rlist_imp_length_le: "<l',l> ∈ rlist(A,r) ==> length(l') ≤ length(l)"*
⟨*proof*⟩

lemma *wf_on_rlist_n:*
"[| n ∈ nat; wf[A](r) |] ==> wf[{l ∈ list(A). length(l) = n}](rlist(A,r))"
⟨*proof*⟩

lemma *list_eq_UN_length: "list(A) = (⋃n∈nat. {l ∈ list(A). length(l) = n})"*
⟨*proof*⟩

lemma *wf_on_rlist: "wf[A](r) ==> wf[list(A)](rlist(A,r))"*
⟨*proof*⟩

lemma *wf_rlist: "wf(r) ==> wf(rlist(field(r),r))"*
⟨*proof*⟩

lemma *well_ord_rlist:*
"well_ord(A,r) ==> well_ord(list(A), rlist(A,r))"
⟨*proof*⟩

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*:

⟦*well_ord(A, r); InfCard(|A|)*⟧ ⟹ *A* × *A* ≈ *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"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *(in Nat_Times_Nat) enum_type [TC,simp]:*
```
    "p ∈ formula ==> enum(fn,p) ∈ nat"
```
⟨*proof*⟩

lemma *(in Nat_Times_Nat) enum_inject [rule_format]:*
```
    "p ∈ formula ==> ∀q∈formula. enum(fn,p) = enum(fn,q) --> p=q"
```
⟨*proof*⟩

lemma *(in Nat_Times_Nat) inj_formula_nat:*
```
    "(λp ∈ formula. enum(fn,p)) ∈ inj(formula, nat)"
```
⟨*proof*⟩

lemma *(in Nat_Times_Nat) well_ord_formula:*
```
    "well_ord(formula, measure(formula, enum(fn)))"
```
⟨*proof*⟩

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* $\lesssim$ *nat"*
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *DPow_imp_ex_DPow_ord:*
```
    "X ∈ DPow(A) ==> ∃k. DPow_ord(fn,r,A,X,k)"
```
⟨*proof*⟩

lemma *(***in** *Nat_Times_Nat) DPow_ord_imp_Ord:*
```
     "[|DPow_ord(fn,r,A,X,k); well_ord(A,r)|] ==> Ord(k)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *(***in** *Nat_Times_Nat) well_ord_DPow_r:*
```
"well_ord(A,r) ==> well_ord(DPow(A), DPow_r(fn,r,A))"
```
⟨*proof*⟩

lemma *(***in** *Nat_Times_Nat) DPow_r_type:*
```
"DPow_r(fn,r,A) ⊆ DPow(A) * DPow(A)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *wf_on_Lset:*
```
"wf[Lset(succ(j))](r(succ(j))) ==> wf[Lset_new(j)](rlimit(i,r))"
```
⟨*proof*⟩

lemma *wf_on_rlimit:*
```
"(∀j<i. wf[Lset(j)](r(j))) ==> wf[Lset(i)](rlimit(i,r))"
```
⟨*proof*⟩

lemma *linear_rlimit:*
```
"[|Limit(i); ∀j<i. linear(Lset(j), r(j)) |]
 ==> linear(Lset(i), rlimit(i,r))"
```
⟨*proof*⟩

lemma *well_ord_rlimit:*
```
"[|Limit(i); ∀j<i. well_ord(Lset(j), r(j)) |]
 ==> well_ord(Lset(i), rlimit(i,r))"
```
⟨*proof*⟩

lemma *rlimit_cong:*
```
"(!!j. j<i ==> r'(j) = r(j)) ==> rlimit(i,r) = rlimit(i,r')"
```
⟨*proof*⟩

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"*
⟨*proof*⟩

lemma *[simp]: "L_r(f, succ(i)) = DPow_r(f, L_r(f,i), Lset(i))"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

lemma *(***in** *Nat_Times_Nat) L_r_type:*
```
"Ord(i) ==> L_r(fn,i) ⊆ Lset(i) * Lset(i)"
```
⟨*proof*⟩

lemma *(***in** *Nat_Times_Nat) well_ord_L_r:*
```
"Ord(i) ==> well_ord(Lset(i), L_r(fn,i))"
```

⟨proof⟩

lemma *well_ord_L_r:*
```
    "Ord(i) ==> ∃r. well_ord(Lset(i), r)"
```
⟨proof⟩

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)"*
⟨proof⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

```
lemma ord_iso_converse1:
     "[| f: ord_iso(A,r,B,s);  <b, f`a>: s;  a:A;  b:B |]
      ==> <converse(f) ` b, a> ∈ r"
⟨proof⟩
```

```
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)))"
⟨proof⟩
```

```
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"
⟨proof⟩
```

```
lemma (in M_ordertype) omap_yields_Ord:
     "[| omap(M,A,r,f); ⟨a,x⟩ ∈ f; M(a); M(x) |]  ==> Ord(x)"
  ⟨proof⟩
```

```
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))))"
⟨proof⟩
```

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)"
```
⟨*proof*⟩

lemma *(in* M_ordertype*)* *Ord_otype:*
```
"[| otype(M,A,r,i); trans[A](r); M(A); M(r); M(i) |] ==> Ord(i)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)⟩)"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

Main result: *om* gives the order-isomorphism ⟨`A, r`⟩ ≅ ⟨`i, Memrel(i)`⟩

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))"
```
⟨*proof*⟩

lemma *(in M_ordertype) obase_exists:*

```
"[| M(A); M(r) |] ==> M(obase(M,A,r))"
```
⟨*proof*⟩

lemma *(in M_ordertype) omap_exists:*

```
"[| M(A); M(r) |] ==> ∃z[M]. omap(M,A,r,z)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)))"
```
⟨*proof*⟩

lemma *(in M_ordertype) relativized_imp_well_ord:*

```
"[| wellordered(M,A,r); M(A); M(r) |] ==> well_ord(A,r)"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

(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)"
```
⟨*proof*⟩

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)"
⟨proof⟩
```

```
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))"
⟨proof⟩
```

```
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)"
⟨proof⟩
```

```
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)"
⟨proof⟩
```

```
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})"
⟨proof⟩
```

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

lemma *(in M_ord_arith) oadd_abs:*
```
"[| M(i); M(j); M(k) |] ==> is_oadd(M,i,j,k) <-> k = i++j"
```
⟨*proof*⟩

lemma *(in M_ord_arith) oadd_closed [intro,simp]:*
```
"[| M(i); M(j) |] ==> M(i++j)"
```
⟨*proof*⟩

16.3.2 Ordinal Multiplication

lemma *omult_eqns_unique:*
```
"[| omult_eqns(i,x,g,z); omult_eqns(i,x,g,z') |] ==> z=z'"
```
⟨*proof*⟩

lemma *omult_eqns_0: "omult_eqns(i,0,g,z) <-> z=0"*
⟨*proof*⟩

lemma *the_omult_eqns_0: "(THE z. omult_eqns(i,0,g,z)) = 0"*
⟨*proof*⟩

lemma *omult_eqns_succ: "omult_eqns(i,succ(j),g,z) <-> Ord(j) & z = g`j
++ i"*
⟨*proof*⟩

lemma *the_omult_eqns_succ:*
```
"Ord(j) ==> (THE z. omult_eqns(i,succ(j),g,z)) = g`j ++ i"
```
⟨*proof*⟩

lemma *omult_eqns_Limit:*
```
"Limit(x) ==> omult_eqns(i,x,g,z) <-> z = ⋃(g``x)"
```
⟨*proof*⟩

lemma *the_omult_eqns_Limit:*
```
"Limit(x) ==> (THE z. omult_eqns(i,x,g,z)) = ⋃(g``x)"
```
⟨*proof*⟩

lemma *omult_eqns_Not: "~ Ord(x) ==> ~ omult_eqns(i,x,g,z)"*
⟨*proof*⟩

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))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

lemma *(in M_ord_arith) exists_omult_fun:*
```
"[| Ord(j);  M(i);  M(j) |] ==> ∃f[M]. is_omult_fun(M,i,succ(j),f)"
```
⟨*proof*⟩

lemma *(in M_ord_arith) is_omult_fun_apply_0:*
```
"[| 0 < j; is_omult_fun(M,i,j,f) |] ==> f`0 = 0"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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"
```
⟨*proof*⟩

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)))"
```

⟨proof⟩

```
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))"
```

⟨proof⟩

```
lemma (in M_wfrank) Ord_wfrank_separation':
     "M(r) ==>
      separation (M, λx.
         ~ (∀f[M]. is_recfun(r^+, x, λx. range, f) --> Ord(range(f))))"
```

⟨proof⟩

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)"
```

⟨*proof*⟩

lemma *(in* `M_wfrank`*)* `M_wellfoundedrank:`

```
    "[| wellfounded(M,r); M(r); M(A) |] ==> M(wellfoundedrank(M,r,A))"
```

⟨*proof*⟩

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))"
```

⟨*proof*⟩

lemma *(in* `M_wfrank`*)* `Ord_range_wellfoundedrank:`

```
    "[| wellfounded(M,r); r ⊆ A*A;  M(r); M(A) |]
     ==> Ord (range(wellfoundedrank(M,r,A)))"
```

⟨*proof*⟩

lemma *(in* `M_wfrank`*)* `function_wellfoundedrank:`

```
    "[| wellfounded(M,r); M(r); M(A)|]
     ==> function(wellfoundedrank(M,r,A))"
```

⟨*proof*⟩

lemma *(in* `M_wfrank`*)* `domain_wellfoundedrank:`

```
    "[| wellfounded(M,r); M(r); M(A)|]
     ==> domain(wellfoundedrank(M,r,A)) = A"
```

⟨*proof*⟩

lemma *(in* `M_wfrank`*)* `wellfoundedrank_type:`

```
    "[| wellfounded(M,r);  M(r); M(A)|]
     ==> wellfoundedrank(M,r,A) ∈ A -> range(wellfoundedrank(M,r,A))"
```

⟨*proof*⟩

lemma *(in* `M_wfrank`*)* `Ord_wellfoundedrank:`

```
    "[| wellfounded(M,r); a ∈ A; r ⊆ A*A;  M(r); M(A) |]
     ==> Ord(wellfoundedrank(M,r,A) ` a)"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

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"
```
⟨*proof*⟩

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))"
```
⟨*proof*⟩

lemma *(in M_wfrank) wellfounded_imp_wf:*
```
"[|wellfounded(M,r); relation(r); M(r)|] ==> wf(r)"
```
⟨*proof*⟩

lemma *(in M_wfrank) wellfounded_on_imp_wf_on:*
```
"[|wellfounded_on(M,A,r); relation(r); M(r); M(A)|] ==> wf[A](r)"
```
⟨*proof*⟩

theorem *(in M_wfrank) wf_abs:*
```
"[|relation(r); M(r)|] ==> wellfounded(M,r) <-> wf(r)"
```
⟨*proof*⟩

theorem *(in M_wfrank) wf_on_abs:*
```
"[|relation(r); M(r); M(A)|] ==> wellfounded_on(M,A,r) <-> wf[A](r)"
```
⟨*proof*⟩

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)]"
```

⟨proof⟩

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)))"
```

⟨proof⟩

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,pa
&
             order_isomorphism(##Lset(i),par,r,x,mx,g)]"
```

⟨proof⟩

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))"
```

⟨proof⟩

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)))]"
```

⟨proof⟩

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))))"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"*
⟨proof⟩

theorem *M_ordertype_L: "PROP M_ordertype(L)"*
⟨proof⟩

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))]"
```
⟨proof⟩

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)))"
```
⟨proof⟩

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)))]"
```
⟨proof⟩

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)))"
```
⟨proof⟩

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))]"
```
⟨proof⟩

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)))"
```
⟨*proof*⟩

17.3.4 Instantiating the locale *M_wfrank*

lemma *M_wfrank_axioms_L: "M_wfrank_axioms(L)"*
⟨*proof*⟩

theorem *M_wfrank_L: "PROP M_wfrank(L)"*
⟨*proof*⟩

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/.