Recursion Theorem

Georgy Dunaev

April 17, 2024

Abstract

This document contains a proof of the recursion theorem. This is a mechanization of the proof of the recursion theorem from the text *Introduction to Set Theory*, by Karel Hrbacek and Thomas Jech. This implementation may be used as the basis for a model of Peano Arithmetic in ZF. While recursion and the natural numbers are already available in ZF, this clean development is much easier to follow.

Contents

1 Recursion Submission **1**

2 Basic Set Theory **1**

3 Compatible set **9**

4 Partial computation **12**

5 Set of functions **13**

6 Lemmas for recursion theorem **19**

7 Recursion theorem **29**

8 Lemmas for addition **30**

9 Definition of addition **32**

1 Recursion Submission

Recursion Theorem is proved in the following document. It also contains the addition on natural numbers. The development is done in the context of Zermelo-Fraenkel set theory.

theory *recursion*
  imports *ZF*
begin

2 Basic Set Theory

Useful lemmas about sets, functions and natural numbers

```
lemma pisubsig : ‹Pi(A,P)⊆Pow(Sigma(A,P))›
proof
  fix x
  assume ‹x ∈ Pi(A,P)›
  hence ‹x ∈ {f∈Pow(Sigma(A,P)). A⊆domain(f) & function(f)}›
    by (unfold Pi-def)
  thus ‹x ∈ Pow(Sigma(A, P))›
    by (rule CollectD1)
qed

lemma apparg:
  fixes f A B
  assumes T0:‹f:A→B›
  assumes T1:‹f ‘ a = b›
  assumes T2:‹a ∈ A›
  shows ‹⟨a, b⟩ ∈ f›
proof(rule iffD2[OF func.apply-iff], rule T0)
  show T:‹a ∈ A ∧ f ‘ a = b›
    by (rule conjI[OF T2 T1])
qed

theorem nat-induct-bound :
  assumes H0:‹P(0)›
  assumes H1:‹!!x. x∈nat ⟹ P(x) ⟹ P(succ(x))›
  shows ‹∀ n∈nat. P(n)›
proof(rule ballI)
  fix n
  assume H2:‹n∈nat›
  show ‹P(n)›
  proof(rule nat-induct[of n])
    from H2 show ‹n∈nat› by assumption
  next
    show ‹P(0)› by (rule H0)
  next
    fix x
    assume H3:‹x∈nat›
    assume H4:‹P(x)›
    show ‹P(succ(x))› by (rule H1[OF H3 H4])
  qed
qed

theorem nat-Tr : ‹∀ n∈nat. m∈n ⟶ m∈nat›
proof(rule nat-induct-bound)
  show ‹m ∈ 0 ⟶ m ∈ nat› by auto
next
  fix x
```

```
  assume H0:‹x ∈ nat›
  assume H1:‹m ∈ x ⟶ m ∈ nat›
  show ‹m ∈ succ(x) ⟶ m ∈ nat›
  proof(rule impI)
    assume H2:‹m∈succ(x)›
    show ‹m ∈ nat›
    proof(rule succE[OF H2])
      assume H3:‹m = x›
      from H0 and H3 show ‹m ∈ nat›
        by auto
    next
      assume H4:‹m ∈ x›
      show ‹m ∈ nat›
        by(rule mp[OF H1 H4])
    qed
  qed
qed


theorem zeroleq : ‹∀ n∈nat. 0∈n ∨ 0=n›
proof(rule ballI)
  fix n
  assume H1:‹n∈nat›
  show ‹0∈n∨0=n›
  proof(rule nat-induct[of n])
    from H1 show ‹n ∈ nat› by assumption
  next
    show ‹0 ∈ 0 ∨ 0 = 0› by (rule disjI2, rule refl)
  next
    fix x
    assume H2:‹x∈nat›
    assume H3:‹ 0 ∈ x ∨ 0 = x›
    show ‹0 ∈ succ(x) ∨ 0 = succ(x)›
    proof(rule disjE[OF H3])
      assume H4:‹0∈x›
      show ‹0 ∈ succ(x) ∨ 0 = succ(x)›
      proof(rule disjI1)
        show ‹0 ∈ succ(x)›
          by (rule succI2[OF H4])
      qed
    next
      assume H4:‹0=x›
      show ‹0 ∈ succ(x) ∨ 0 = succ(x)›
      proof(rule disjI1)
        have q:‹x ∈ succ(x)› by auto
        from q and H4 show ‹0 ∈ succ(x)› by auto
      qed
    qed
  qed
```

```
qed

theorem JH2-1ii : ‹m∈succ(n) ⟹ m∈n∨m=n›
  by auto

theorem nat-transitive:‹∀ n∈nat. ∀ k. ∀ m.  k ∈ m ∧ m ∈ n ⟶ k ∈ n›
proof(rule nat-induct-bound)
  show ‹∀ k. ∀ m. k ∈ m ∧ m ∈ 0 ⟶ k ∈ 0›
  proof(rule allI, rule allI, rule impI)
    fix k m
    assume H:‹k ∈ m ∧ m ∈ 0›
    then have H:‹m ∈ 0› by auto
    then show ‹k ∈ 0› by auto
  qed
next
  fix n
  assume H0:‹n ∈ nat›
  assume H1:‹∀ k.
          ∀ m.
             k ∈ m ∧ m ∈ n ⟶
             k ∈ n›
  show ‹∀ k. ∀ m.
             k ∈ m ∧
             m ∈ succ(n) ⟶
             k ∈ succ(n)›
  proof(rule allI, rule allI, rule impI)
    fix k m
    assume H4:‹k ∈ m ∧ m ∈ succ(n)›
    hence H4':‹m ∈ succ(n)› by (rule conjunct2)
    hence H4'':‹m∈n ∨ m=n› by (rule succE, auto)
    from H4 have Q:‹k ∈ m› by (rule conjunct1)
    have H1S:‹∀ m. k ∈ m ∧ m ∈ n ⟶ k ∈ n›
      by (rule spec[OF H1])
    have H1S:‹k ∈ m ∧ m ∈ n ⟶ k ∈ n›
      by (rule spec[OF H1S])
    show ‹k ∈ succ(n)›
    proof(rule disjE[OF H4'])
      assume L:‹m∈n›
      from Q and L have QL:‹k ∈ m ∧ m ∈ n› by auto
      have G:‹k ∈ n› by (rule mp [OF H1S QL])
      show ‹k ∈ succ(n)›
        by (rule succI2[OF G])
    next
      assume L:‹m=n›
      from Q have F:‹k ∈ succ(m)› by auto
      from L and Q show ‹k ∈ succ(n)› by auto
    qed
  qed
qed
```

```
theorem nat-xninx : ‹∀ n∈nat. ¬(n∈n)›
proof(rule nat-induct-bound)
  show ‹0∉0›
    by auto
next
  fix x
  assume H0:‹x∈nat›
  assume H1:‹x∉x›
  show ‹succ(x) ∉ succ(x)›
  proof(rule contrapos[OF H1])
    assume Q:‹succ(x) ∈ succ(x)›
    have D:‹succ(x)∈x ∨ succ(x)=x›
      by (rule JH2-1ii[OF Q])
    show ‹x∈x›
    proof(rule disjE[OF D])
      assume Y1:‹succ(x)∈x›
      have U:‹x∈succ(x)› by (rule succI1)
      have T:‹x ∈ succ(x) ∧ succ(x) ∈ x ⟶ x ∈ x›
        by (rule spec[OF spec[OF bspec[OF nat-transitive H0]]])
      have R:‹x ∈ succ(x) ∧ succ(x) ∈ x›
        by (rule conjI[OF U Y1])
      show ‹x∈x›
        by (rule mp[OF T R])
    next
      assume Y1:‹succ(x)=x›
      show ‹x∈x›
        by (rule subst[OF Y1], rule Q)
    qed
  qed
qed

theorem nat-asym : ‹∀ n∈nat. ∀ m. ¬(n∈m ∧ m∈n)›
proof(rule ballI, rule allI)
  fix n m
  assume H0:‹n ∈ nat›
  have Q:‹¬(n∈n)›
    by(rule bspec[OF nat-xninx H0])
  show ‹¬ (n ∈ m ∧ m ∈ n)›
  proof(rule contrapos[OF Q])
    assume W:‹(n ∈ m ∧ m ∈ n)›
    show ‹n∈n›
      by (rule mp[OF spec[OF spec[OF bspec[OF nat-transitive H0]]] W])
  qed
qed

theorem zerolesucc :‹∀ n∈nat. 0 ∈ succ(n)›
proof(rule nat-induct-bound)
  show ‹0∈1›
```

```
    by auto
next
  fix x
  assume H0:‹x∈nat›
  assume H1:‹0∈succ(x)›
  show ‹0∈succ(succ(x))›
  proof
    assume J:‹0 ∉ succ(x)›
    show ‹0 = succ(x)›
      by(rule notE[OF J H1])
  qed
qed

theorem succ-le : ‹∀ n∈nat. succ(m)∈succ(n) ⟶ m∈n›
proof(rule nat-induct-bound)
  show ‹ succ(m) ∈ 1 ⟶ m ∈ 0›
    by blast
next
  fix x
  assume H0:‹x ∈ nat›
  assume H1:‹succ(m) ∈ succ(x) ⟶ m ∈ x›
  show ‹ succ(m) ∈
            succ(succ(x)) ⟶
            m ∈ succ(x)›
  proof(rule impI)
    assume J0:‹succ(m) ∈ succ(succ(x))›
    show ‹m ∈ succ(x)›
    proof(rule succE[OF J0])
      assume R:‹succ(m) = succ(x)›
      hence R:‹m=x› by (rule upair.succ-inject)
      from R and succI1 show ‹m ∈ succ(x)› by auto
    next
      assume R:‹succ(m) ∈ succ(x)›
      have R:‹m∈x› by (rule mp[OF H1 R])
      then show ‹m ∈ succ(x)› by auto
    qed
  qed
qed

theorem succ-le2 : ‹∀ n∈nat. ∀ m. succ(m)∈succ(n) ⟶ m∈n›
proof
  fix n
  assume H:‹n∈nat›
  show ‹∀ m. succ(m) ∈ succ(n) ⟶ m ∈ n›
  proof
    fix m
    from succ-le and H show ‹succ(m) ∈ succ(n) ⟶ m ∈ n› by auto
  qed
qed
```

```
theorem le-succ : ‹∀ n∈nat. m∈n ⟶ succ(m)∈succ(n)›
proof(rule nat-induct-bound)
  show ‹m ∈ 0 ⟶ succ(m) ∈ 1›
    by auto
next
  fix x
  assume H0:‹x∈nat›
  assume H1:‹m ∈ x ⟶ succ(m) ∈ succ(x)›
  show ‹m ∈ succ(x) ⟶
          succ(m) ∈ succ(succ(x))›
  proof(rule impI)
    assume HR1:‹m∈succ(x)›
    show ‹succ(m) ∈ succ(succ(x))›
    proof(rule succE[OF HR1])
      assume Q:‹m = x›
      from Q show ‹succ(m) ∈ succ(succ(x))›
        by auto
    next
      assume Q:‹m ∈ x›
      have Q:‹succ(m) ∈ succ(x)›
        by (rule mp[OF H1 Q])
      from Q show ‹succ(m) ∈ succ(succ(x))›
        by (rule succI2)
    qed
  qed
qed

theorem nat-linord:‹∀ n∈nat. ∀ m∈nat. m∈n∨m=n∨n∈m›
proof(rule ballI)
  fix n
  assume H1:‹n∈nat›
  show ‹∀ m∈nat. m ∈ n ∨ m = n ∨ n ∈ m›
  proof(rule nat-induct[of n])
    from H1 show ‹n∈nat› by assumption
  next
    show ‹∀ m∈nat. m ∈ 0 ∨ m = 0 ∨ 0 ∈ m›
    proof
      fix m
      assume J:‹m∈nat›
      show ‹ m ∈ 0 ∨ m = 0 ∨ 0 ∈ m›
      proof(rule disjI2)
        have Q:‹0∈m∨0=m› by (rule bspec[OF zeroleq J])
        show ‹m = 0 ∨ 0 ∈ m›
          by (rule disjE[OF Q], auto)
      qed
    qed
  next
    fix x
```

```
assume K:‹x∈nat›
assume M:‹∀ m∈nat. m ∈ x ∨ m = x ∨ x ∈ m›
show ‹∀ m∈nat.
        m ∈ succ(x) ∨
        m = succ(x) ∨
        succ(x) ∈ m›
proof(rule nat-induct-bound)
  show ‹0 ∈ succ(x) ∨  0 = succ(x) ∨ succ(x) ∈ 0›
  proof(rule disjI1)
    show ‹0 ∈ succ(x)›
      by (rule bspec[OF zerolesucc K])
  qed
next
  fix y
  assume H0:‹y ∈ nat›
  assume H1:‹y ∈ succ(x) ∨ y = succ(x) ∨ succ(x) ∈ y›
  show ‹succ(y) ∈ succ(x) ∨
        succ(y) = succ(x) ∨
        succ(x) ∈ succ(y)›
  proof(rule disjE[OF H1])
    assume W:‹y∈succ(x)›
    show ‹succ(y) ∈ succ(x) ∨
          succ(y) = succ(x) ∨
          succ(x) ∈ succ(y)›
    proof(rule succE[OF W])
      assume G:‹y=x›
      show ‹succ(y) ∈ succ(x) ∨
succ(y) = succ(x) ∨
succ(x) ∈ succ(y)›
        by (rule disjI2, rule disjI1, rule subst[OF G], rule refl)
    next
      assume G:‹y ∈ x›
      have R:‹succ(y) ∈ succ(x)›
        by (rule mp[OF bspec[OF le-succ K] G])
      show ‹succ(y) ∈ succ(x) ∨
       succ(y) = succ(x) ∨
       succ(x) ∈ succ(y)›
        by(rule disjI1, rule R)
    qed
  next
    assume W:‹y = succ(x) ∨ succ(x) ∈ y›
    show ‹succ(y) ∈ succ(x) ∨
          succ(y) = succ(x) ∨
          succ(x) ∈ succ(y)›
    proof(rule disjE[OF W])
      assume W:‹y=succ(x)›
      show ‹succ(y) ∈ succ(x) ∨
          succ(y) = succ(x) ∨
          succ(x) ∈ succ(y)›
```

```
          by (rule disjI2, rule disjI2, rule subst[OF W], rule succI1)
        next
          assume W:‹succ(x)∈y›
          show ‹succ(y) ∈ succ(x) ∨
              succ(y) = succ(x) ∨
              succ(x) ∈ succ(y)›
          by (rule disjI2, rule disjI2, rule succI2[OF W])
        qed
      qed
    qed
  qed
qed

lemma tgb:
  assumes knat: ‹k∈nat›
  assumes D: ‹t ∈ k → A›
  shows  ‹t ∈ Pow(nat × A)›
proof −
  from D
  have q:‹t∈{t∈Pow(Sigma(k,%-.A)). k⊆domain(t) & function(t)}›
    by(unfold Pi-def)
  have J:‹t ∈ Pow(k × A)›
    by (rule CollectD1[OF q])
  have G:‹k × A ⊆ nat × A›
  proof(rule func.Sigma-mono)
    from knat
    show ‹k⊆nat›
      by (rule QUniv.naturals-subset-nat)
  next
    show ‹⋀x. x ∈ k ⟹ A ⊆ A›
      by auto
  qed
  show ‹t ∈ Pow(nat × A)›
    by (rule subsetD, rule func.Pow-mono[OF G], rule J)
qed
```

3 Compatible set

Union of compatible set of functions is a function.

```
definition compat :: ‹[i,i]⇒o›
  where compat(f1,f2) == ∀x.∀y1.∀y2.⟨x,y1⟩ ∈ f1 ∧ ⟨x,y2⟩ ∈ f2 ⟶ y1=y2

lemma compatI [intro]:
  assumes H:‹⋀x y1 y2.⟦⟨x,y1⟩ ∈ f1; ⟨x,y2⟩ ∈ f2⟧⟹y1=y2›
  shows ‹compat(f1,f2)›
proof(unfold compat-def)
  show ‹∀x y1 y2. ⟨x, y1⟩ ∈ f1 ∧ ⟨x, y2⟩ ∈ f2 ⟶ y1 = y2›
  proof(rule allI | rule impI)+
```

```
    fix x y1 y2
    assume K:‹⟨x, y1⟩ ∈ f1 ∧ ⟨x, y2⟩ ∈ f2›
    have K1:‹⟨x, y1⟩ ∈ f1› by (rule conjunct1[OF K])
    have K2:‹⟨x, y2⟩ ∈ f2› by (rule conjunct2[OF K])
    show ‹y1 = y2› by (rule H[OF K1 K2])
  qed
qed

lemma compatD:
  assumes H: ‹compat(f1,f2)›
  shows ‹⋀x y1 y2.⟦⟨x,y1⟩ ∈ f1; ⟨x,y2⟩ ∈ f2⟧⟹y1=y2›
proof −
  fix x y1 y2
  assume Q1:‹⟨x, y1⟩ ∈ f1›
  assume Q2:‹⟨x, y2⟩ ∈ f2›
  from H have H:‹∀x y1 y2. ⟨x, y1⟩ ∈ f1 ∧ ⟨x, y2⟩ ∈ f2 ⟶ y1 = y2›
    by (unfold compat-def)
  show ‹y1=y2›
  proof(rule mp[OF spec[OF spec[OF spec[OF H]]]])
    show ‹⟨x, y1⟩ ∈ f1 ∧ ⟨x, y2⟩ ∈ f2›
      by(rule conjI[OF Q1 Q2])
  qed
qed

lemma compatE:
  assumes H: ‹compat(f1,f2)›
  and W:‹(⋀x y1 y2.⟦⟨x,y1⟩ ∈ f1; ⟨x,y2⟩ ∈ f2⟧⟹y1=y2) ⟹ E›
shows ‹E›
  by (rule W, rule compatD[OF H], assumption+)


definition compatset :: ‹i⇒o›
  where compatset(S) == ∀f1∈S.∀f2∈S. compat(f1,f2)

lemma compatsetI [intro] :
  assumes 1:‹⋀f1 f2. ⟦f1∈S;f2∈S⟧ ⟹ compat(f1,f2)›
  shows ‹compatset(S)›
  by (unfold compatset-def, rule ballI, rule ballI, rule 1, assumption+)

lemma compatsetD:
  assumes H: ‹compatset(S)›
  shows ‹⋀f1 f2.⟦f1∈S; f2∈S⟧⟹compat(f1,f2)›
proof −
  fix f1 f2
  assume H1:‹f1∈S›
  assume H2:‹f2∈S›
  from H have H:‹∀f1∈S.∀f2∈S. compat(f1,f2)›
    by (unfold compatset-def)
  show ‹compat(f1,f2)›
```

```
    by (rule bspec[OF bspec[OF H H1] H2])
qed

lemma compatsetE:
  assumes H: ‹compatset(S)›
  and W:‹(⋀f1 f2.⟦f1∈S; f2∈S⟧⟹compat(f1,f2)) ⟹ E›
shows ‹E›
  by (rule W, rule compatsetD[OF H], assumption+)

theorem upairI1 : ‹a ∈ {a, b}›
proof
  assume ‹a ∉ {b}›
  show ‹a = a› by (rule refl)
qed

theorem upairI2 : ‹b ∈ {a, b}›
proof
  assume H:‹b ∉ {b}›
  have Y:‹b ∈ {b}› by (rule upair.singletonI)
  show ‹b = a› by (rule notE[OF H Y])
qed

theorem sinup : ‹{x} ∈ ⟨x, xa⟩›
proof (unfold Pair-def)
  show ‹{x} ∈ {{x, x}, {x, xa}}›
  proof (rule IFOL.subst)
    show ‹{x} ∈ {{x},{x,xa}}›
      by (rule upairI1)
  next
    show ‹{{x}, {x, xa}} = {{x, x}, {x, xa}}›
      by blast
  qed
qed

theorem compatsetunionfun :
  fixes S
  assumes H0:‹compatset(S)›
  shows ‹function(⋃S)›
proof(unfold function-def)
  show ‹ ∀x y1. ⟨x, y1⟩ ∈ ⋃S ⟶
          (∀y2. ⟨x, y2⟩ ∈ ⋃S ⟶ y1 = y2)›
  proof(rule allI, rule allI, rule impI, rule allI, rule impI)
    fix x y1 y2
    assume F1:‹⟨x, y1⟩ ∈ ⋃S›
    assume F2:‹⟨x, y2⟩ ∈ ⋃S›
    show ‹y1=y2›
    proof(rule UnionE[OF F1], rule UnionE[OF F2])
      fix f1 f2
      assume J1:‹⟨x, y1⟩ ∈ f1›
```

```
      assume J2:‹⟨x, y2⟩ ∈ f2›
      assume K1:‹f1 ∈ S›
      assume K2:‹f2 ∈ S›
      have R:‹compat(f1,f2)›
        by (rule compatsetD[OF H0 K1 K2])
      show ‹y1=y2›
        by(rule compatD[OF R J1 J2])
    qed
  qed
qed

theorem mkel :
  assumes 1:‹A›
  assumes 2:‹A⟹B›
  shows ‹B›
  by (rule 2, rule 1)

theorem valofunion :
  fixes S
  assumes H0:‹compatset(S)›
  assumes W:‹f∈S›
  assumes Q:‹f:A→B›
  assumes T:‹a∈A›
  assumes P:‹f ‘ a = v›
  shows N:‹(⋃S)‘a = v›
proof −
  have K:‹⟨a, v⟩ ∈ f›
    by (rule apparg[OF Q P T])
  show N:‹(⋃S)‘a = v›
  proof(rule function-apply-equality)
    show ‹function(⋃S)›
      by(rule compatsetunionfun[OF H0])
  next
    show ‹⟨a, v⟩ ∈ ⋃S›
      by(rule UnionI[OF W K ])
  qed
qed
```

4 Partial computation

```
definition satpc :: ‹[i,i,i] ⇒ o ›
  where ‹satpc(t,α,g) == ∀ n ∈ α . t‘succ(n) = g ‘ <t‘n, n>›
```

m-step computation based on a and g

```
definition partcomp :: ‹[i,i,i,i,i]⇒o›
  where ‹partcomp(A,t,m,a,g) == (t:succ(m)→A) ∧ (t‘0=a) ∧ satpc(t,m,g)›

lemma partcompI [intro]:
  assumes H1:‹(t:succ(m)→A)›
```

```
  assumes H2:‹(t‘0=a)›
  assumes H3:‹satpc(t,m,g)›
  shows ‹partcomp(A,t,m,a,g)›
proof (unfold partcomp-def, auto)
  show ‹t ∈ succ(m) → A› by (rule H1)
  show ‹(t‘0=a)› by (rule H2)
  show ‹satpc(t,m,g)› by (rule H3)
qed

lemma partcompD1: ‹partcomp(A,t,m,a,g) ⟹ t ∈ succ(m) → A›
  by (unfold partcomp-def, auto)

lemma partcompD2: ‹partcomp(A,t,m,a,g) ⟹ (t‘0=a)›
 by (unfold partcomp-def, auto)

lemma partcompD3: ‹partcomp(A,t,m,a,g) ⟹ satpc(t,m,g)›
  by (unfold partcomp-def, auto)

lemma partcompE [elim] :
  assumes 1:‹partcomp(A,t,m,a,g)›
    and 2:‹⟦(t:succ(m)→A) ; (t‘0=a) ; satpc(t,m,g)⟧ ⟹ E›
  shows ‹E›
 by (rule 2, rule partcompD1[OF 1], rule partcompD2[OF 1], rule partcompD3[OF
1])
```

If we add ordered pair in the middle of partial computation then it will not change.

```
lemma addmiddle:

  assumes mnat:‹m∈nat›
  assumes F:‹partcomp(A,t,m,a,g)›
  assumes xinm:‹x∈m›
  shows ‹cons(⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩, t) = t›
proof(rule partcompE[OF F])
  assume F1:‹t ∈ succ(m) → A›
  assume F2:‹t ‘ 0 = a›
  assume F3:‹satpc(t, m, g)›
  from F3
  have W:‹∀ n∈m. t ‘ succ(n) = g ‘ ⟨t ‘ n, n⟩›
    by (unfold satpc-def)
  have U:‹t ‘ succ(x) = g ‘ ⟨t ‘ x, x⟩›
    by (rule bspec[OF W xinm])
  have E:‹⟨succ(x), (g ‘ ⟨t ‘ x, x⟩)⟩ ∈ t›
  proof(rule apparg[OF F1 U])
    show ‹succ(x) ∈ succ(m)›
      by(rule mp[OF bspec[OF le-succ mnat] xinm])
  qed
  show ?thesis
    by (rule equalities.cons-absorb[OF E])
```

qed

5 Set of functions

It is denoted as F on page 48 in "Introduction to Set Theory".

```
definition pcs :: ‹[i,i,i]⇒i›
  where ‹pcs(A,a,g) == {t∈Pow(nat∗A). ∃ m∈nat. partcomp(A,t,m,a,g)}›

lemma pcs-uniq :
  assumes F1:‹m1∈nat›
  assumes F2:‹m2∈nat›
  assumes H1: ‹partcomp(A,f1,m1,a,g)›
  assumes H2: ‹partcomp(A,f2,m2,a,g)›
  shows ‹∀ n∈nat. n∈succ(m1) ∧ n∈succ(m2) ⟶ f1‘n = f2‘n›
proof(rule partcompE[OF H1], rule partcompE[OF H2])
  assume H11:‹f1 ∈ succ(m1) → A›
  assume H12:‹f1 ‘ 0 = a ›
  assume H13:‹satpc(f1, m1, g)›
  assume H21:‹f2 ∈ succ(m2) → A›
  assume H22:‹f2 ‘ 0 = a›
  assume H23:‹satpc(f2, m2, g)›
  show ‹∀ n∈nat. n∈succ(m1) ∧ n∈succ(m2) ⟶ f1‘n = f2‘n›
proof(rule nat-induct-bound)
  from H12 and H22
  show ‹0∈succ(m1) ∧ 0∈succ(m2) ⟶ f1 ‘ 0 = f2 ‘ 0›
    by auto
next
  fix x
  assume J0:‹x∈nat›
  assume J1:‹x ∈ succ(m1) ∧ x ∈ succ(m2) ⟶ f1 ‘ x = f2 ‘ x›
  from H13 have G1:‹∀ n ∈ m1 . f1‘succ(n) = g ‘ <f1‘n, n>›
    by (unfold satpc-def, auto)
  from H23 have G2:‹∀ n ∈ m2 . f2‘succ(n) = g ‘ <f2‘n, n>›
    by (unfold satpc-def, auto)
  show ‹succ(x) ∈ succ(m1) ∧ succ(x) ∈ succ(m2) ⟶
        f1 ‘ succ(x) = f2 ‘ succ(x)›
  proof
    assume K:‹succ(x) ∈ succ(m1) ∧ succ(x) ∈ succ(m2)›
    from K have K1:‹succ(x) ∈ succ(m1)› by auto
    from K have K2:‹succ(x) ∈ succ(m2)› by auto
    have K1':‹x ∈ m1› by (rule mp[OF bspec[OF succ-le F1] K1])
    have K2':‹x ∈ m2› by (rule mp[OF bspec[OF succ-le F2] K2])
    have U1:‹x∈succ(m1)›
      by (rule Nat.succ-in-naturalD[OF K1 Nat.nat-succI[OF F1]])
    have U2:‹x∈succ(m2)›
      by (rule Nat.succ-in-naturalD[OF K2 Nat.nat-succI[OF F2]])
    have Y1:‹f1‘succ(x) = g ‘ <f1‘x, x>›
      by (rule bspec[OF G1 K1'])
```

```
    have Y2:‹f2‘succ(x) = g ‘ <f2‘x, x>›
      by (rule bspec[OF G2 K2'])
    have ‹f1 ‘ x = f2 ‘ x›
      by(rule mp[OF J1 conjI[OF U1 U2]])
    then have Y:‹g ‘ <f1‘x, x> = g ‘ <f2‘x, x>› by auto
    from Y1 and Y2 and Y
    show ‹f1 ‘ succ(x) = f2 ‘ succ(x)›
      by auto
  qed
qed
qed

lemma domainsubsetfunc :
  assumes Q:‹f1⊆f2›
  shows ‹domain(f1)⊆domain(f2)›
proof
  fix x
  assume H:‹x ∈ domain(f1)›
  show ‹x ∈ domain(f2)›
  proof(rule domainE[OF H])
    fix y
    assume W:‹⟨x, y⟩ ∈ f1›
    have ‹⟨x, y⟩ ∈ f2›
      by(rule subsetD[OF Q W])
    then show ‹x ∈ domain(f2)›
      by(rule domainI)
  qed
qed

lemma natdomfunc:
  assumes 1:‹q∈A›
  assumes J0:‹f1 ∈ Pow(nat × A)›
  assumes U:‹m1 ∈ domain(f1)›
  shows ‹m1∈nat›
proof −
  from J0 have J0 : ‹f1 ⊆ nat × A›
    by auto
  have J0:‹domain(f1) ⊆ domain(nat × A)›
    by(rule func.domain-mono[OF J0])
  have F:‹m1 ∈ domain(nat × A)›
    by(rule subsetD[OF J0 U])
  have R:‹domain(nat × A) = nat›
    by (rule equalities.domain-of-prod[OF 1])
  show ‹m1 ∈ nat›
    by(rule subst[OF R], rule F)
qed

lemma pcs-lem :
  assumes 1:‹q∈A›
```

```
  shows ‹compatset(pcs(A, a, g))›
proof
  fix f1 f2
  assume H1:‹f1 ∈ pcs(A, a, g)›
  then have H1':‹f1 ∈ {t∈Pow(nat∗A). ∃ m∈nat. partcomp(A,t,m,a,g)}› by (unfold
pcs-def)
  hence H1'A:‹f1 ∈ Pow(nat∗A)› by auto
  hence H1'A:‹f1 ⊆ (nat∗A)› by auto
  assume H2:‹f2 ∈ pcs(A, a, g)›
  then have H2':‹f2 ∈ {t∈Pow(nat∗A). ∃ m∈nat. partcomp(A,t,m,a,g)}› by (unfold
pcs-def)
  show ‹compat(f1, f2)›
  proof(rule compatI)
    fix x y1 y2
    assume P1:‹⟨x, y1⟩ ∈ f1›
    assume P2:‹⟨x, y2⟩ ∈ f2›
    show ‹y1 = y2›
    proof(rule CollectE[OF H1'], rule CollectE[OF H2'])
      assume J0:‹f1 ∈ Pow(nat × A)›
      assume J1:‹f2 ∈ Pow(nat × A)›
      assume J2:‹∃ m∈nat. partcomp(A, f1, m, a, g)›
      assume J3:‹∃ m∈nat. partcomp(A, f2, m, a, g)›
      show ‹y1 = y2›
      proof(rule bexE[OF J2], rule bexE[OF J3])
        fix m1 m2
        assume K1:‹partcomp(A, f1, m1, a, g)›
        assume K2:‹partcomp(A, f2, m2, a, g)›
        hence K2':‹(f2:succ(m2)→A) ∧ (f2‘0=a) ∧ satpc(f2,m2,g)›
          by (unfold partcomp-def)
        from K1 have K1'A:‹(f1:succ(m1)→A)› by (rule partcompD1)
        from K2' have K2'A:‹(f2:succ(m2)→A)› by auto
        from K1'A have K1'AD:‹domain(f1) = succ(m1)›
          by(rule domain-of-fun)
        from K2'A have K2'AD:‹domain(f2) = succ(m2)›
          by(rule domain-of-fun)
        have L1:‹f1‘x=y1›
          by (rule func.apply-equality[OF P1], rule K1'A)
        have L2:‹f2‘x=y2›
          by(rule func.apply-equality[OF P2], rule K2'A)
        have m1nat:‹m1∈nat›
        proof(rule natdomfunc[OF 1 J0])
          show ‹m1 ∈ domain(f1)›
            by (rule ssubst[OF K1'AD], auto)
        qed
        have m2nat:‹m2∈nat›
        proof(rule natdomfunc[OF 1 J1])
          show ‹m2 ∈ domain(f2)›
            by (rule ssubst[OF K2'AD], auto)
        qed
```

```
        have G1:‹⟨x, y1⟩ ∈ (nat∗A)›
          by(rule subsetD[OF H1'A P1])
        have KK:‹x∈nat›
          by(rule SigmaE[OF G1], auto)

        have W:‹f1‘x=f2‘x›
        proof(rule mp[OF bspec[OF pcs-uniq KK] ])
          show ‹m1 ∈ nat›
            by (rule m1nat)
        next
          show ‹m2 ∈ nat›
            by (rule m2nat)
        next
          show ‹partcomp(A, f1, m1, a, g)›
            by (rule K1)
        next
          show ‹partcomp(A, f2, m2, a, g)›
            by (rule K2)
        next

          have U1:‹x ∈ succ(m1)›
            by (rule func.domain-type[OF P1 K1'A])
          have U2:‹x ∈ succ(m2)›
            by (rule func.domain-type[OF P2 K2'A])
          show ‹x ∈ succ(m1) ∧ x ∈ succ(m2)›
            by (rule conjI[OF U1 U2])
        qed
        from L1 and W and L2
        show ‹y1 = y2› by auto
      qed
    qed
  qed
qed

theorem fuissu : ‹f ∈ X -> Y ⟹ f ⊆ X×Y›
proof
  fix w
  assume H1 : ‹f ∈ X -> Y›
  then have J1:‹f ∈ {q∈Pow(Sigma(X,λ-. Y)). X⊆domain(q) & function(q)}›
    by (unfold Pi-def)
  then have J2:‹f ∈ Pow(Sigma(X,λ-. Y))›
    by auto
  then have J3:‹f ⊆ Sigma(X,λ-. Y)›
    by auto
  assume H2 : ‹w ∈ f›
  from J3 and H2 have ‹w∈Sigma(X,λ-. Y)›
    by auto
  then have J4:‹w ∈ (⋃x∈X. (⋃y∈Y. {⟨x,y⟩}))›
    by auto
```

```
  show ‹w ∈ X∗Y›
  proof (rule UN-E[OF J4])
    fix x
    assume V1:‹x ∈ X›
    assume V2:‹w ∈ (⋃y∈Y. {⟨x, y⟩})›
    show ‹w ∈ X × Y›
    proof (rule UN-E[OF V2])
      fix y
      assume V3:‹y ∈ Y›
      assume V4:‹w ∈ {⟨x, y⟩}›
      then have V4:‹w = ⟨x, y⟩›
        by auto
      have v5:‹⟨x, y⟩ ∈ Sigma(X,λ-.Y)›
      proof(rule SigmaI)
        show ‹x ∈ X› by (rule V1)
      next
        show ‹y ∈ Y› by (rule V3)
      qed
      then have V5:‹⟨x, y⟩ ∈ X∗Y›
        by auto
      from V4 and V5 show ‹w ∈ X × Y› by auto
    qed
  qed
qed

theorem recuniq :
  fixes f
  assumes H0:‹f ∈ nat −> A ∧ f ‘ 0 = a ∧ satpc(f, nat, g)›
  fixes t
  assumes H1:‹t ∈ nat −> A ∧ t ‘ 0 = a ∧ satpc(t, nat, g)›
  fixes x
  shows ‹f=t›
proof −
  from H0 have H02:‹∀ n ∈ nat. f‘succ(n) = g ‘ <(f‘n), n>› by (unfold satpc-def,
auto)
  from H0 have H01:‹f ‘ 0 = a› by auto
  from H0 have H00:‹f ∈ nat −> A› by auto
  from H1 have H12:‹∀ n ∈ nat. t‘succ(n) = g ‘ <(t‘n), n>› by (unfold satpc-def,
auto)
  from H1 have H11:‹t ‘ 0 = a› by auto
  from H1 have H10:‹t ∈ nat −> A› by auto
  show ‹f=t›
  proof (rule fun-extension[OF H00 H10])
    fix x
    assume K: ‹x ∈ nat›
    show ‹(f ‘ x) = (t ‘ x)›
    proof(rule nat-induct[of x])
      show ‹x ∈ nat› by (rule K)
    next
```

```
      from H01 and H11 show ‹f ‘ 0 = t ‘ 0›
        by auto
    next
      fix x
      assume A:‹x∈nat›
      assume B:‹f‘x = t‘x›
      show ‹f ‘ succ(x) = t ‘ succ(x)›
      proof −
        from H02 and A have H02′:‹f‘succ(x) = g ‘ <(f‘x), x>›
          by (rule bspec)
        from H12 and A have H12′:‹t‘succ(x) = g ‘ <(t‘x), x>›
          by (rule bspec)
        from B and H12′ have H12″:‹t‘succ(x) = g ‘ <(f‘x), x>› by auto
        from H12″ and H02′ show ‹f ‘ succ(x) = t ‘ succ(x)› by auto
      qed
    qed
  qed
qed
```

6 Lemmas for recursion theorem

```
locale recthm =
  fixes A :: i
    and a :: i
    and g :: i
  assumes hyp1 : ‹a ∈ A›
    and hyp2 : ‹g : ((A∗nat)→A)›
begin

lemma l3:‹function(⋃pcs(A, a, g))›
  by (rule compatsetunionfun, rule pcs-lem, rule hyp1)

lemma l1 : ‹⋃pcs(A, a, g) ⊆ nat × A›
proof
  fix x
  assume H:‹x ∈ ⋃pcs(A, a, g)›
  hence  H:‹x ∈ ⋃{t∈Pow(nat∗A). ∃m∈nat. partcomp(A,t,m,a,g)}›
    by (unfold pcs-def)
  show ‹x ∈ nat × A›
  proof(rule UnionE[OF H])
    fix B
    assume J1:‹x∈B›
    assume J2:‹B ∈ {t ∈ Pow(nat × A) .
          ∃m∈nat. partcomp(A, t, m, a, g)}›
    hence J2:‹B ∈ Pow(nat × A)› by auto
    hence J2:‹B ⊆ nat × A› by auto
    from J1 and J2 show ‹x ∈ nat × A›
      by auto
  qed
```

```
qed

lemma le1:
  assumes H:‹x∈1›
  shows ‹x=0›
proof
  show ‹x ⊆ 0›
  proof
    fix z
    assume J:‹z∈x›
    show ‹z∈0›
    proof(rule succE[OF H])
      assume J:‹x∈0›
      show ‹z∈0›
        by (rule notE[OF not-mem-empty J])
    next
      assume K:‹x=0›
      from J and K show ‹z∈0›
        by auto
    qed
  qed
next
  show ‹0 ⊆ x› by auto
qed

lemma lsinglfun : ‹function({⟨0, a⟩})›
proof(unfold function-def)
  show ‹ ∀x y. ⟨x, y⟩ ∈ {⟨0, a⟩} ⟶
          (∀y'. ⟨x, y'⟩ ∈ {⟨0, a⟩} ⟶
               y = y')›
  proof(rule allI,rule allI,rule impI,rule allI,rule impI)
    fix x y y'
    assume H0:‹⟨x, y⟩ ∈ {⟨0, a⟩}›
    assume H1:‹⟨x, y'⟩ ∈ {⟨0, a⟩}›
    show ‹y = y'›
    proof(rule upair.singletonE[OF H0],rule upair.singletonE[OF H1])
      assume H0:‹⟨x, y⟩ = ⟨0, a⟩›
      assume H1:‹⟨x, y'⟩ = ⟨0, a⟩›
      from H0 and H1 have H:‹⟨x, y⟩ = ⟨x, y'⟩› by auto
      then show ‹y = y'› by auto
    qed
  qed
qed

lemma singlsatpc:‹satpc({⟨0, a⟩}, 0, g)›
proof(unfold satpc-def)
  show ‹∀n∈0. {⟨0, a⟩} ` succ(n) =
          g ` ⟨{⟨0, a⟩} ` n, n⟩›
    by auto
```

```
qed

lemma zerostep :
  shows ‹partcomp(A, {⟨0, a⟩}, 0, a, g)›
proof(unfold partcomp-def)
  show ‹{⟨0, a⟩} ∈ 1 -> A ∧ {⟨0, a⟩} ` 0 = a ∧ satpc({⟨0, a⟩}, 0, g)›
  proof
    show ‹{⟨0, a⟩} ∈ 1 -> A›
    proof (unfold Pi-def)
      show ‹{⟨0, a⟩} ∈ {f ∈ Pow(1 × A) . 1 ⊆ domain(f) ∧ function(f)}›
      proof
        show ‹{⟨0, a⟩} ∈ Pow(1 × A)›
        proof(rule PowI, rule equalities.singleton-subsetI)
          show ‹⟨0, a⟩ ∈ 1 × A›
          proof
            show ‹0 ∈ 1› by auto
          next
            show ‹a ∈ A› by (rule hyp1)
          qed
        qed
      next
        show ‹1 ⊆ domain({⟨0, a⟩}) ∧ function({⟨0, a⟩})›
        proof
          show ‹1 ⊆ domain({⟨0, a⟩})›
          proof
            fix x
            assume W:‹x∈1›
            from W have W:‹x=0› by (rule le1)
            have Y:‹0∈domain({⟨0, a⟩})›
              by auto
            from W and Y
            show ‹x∈domain({⟨0, a⟩})›
              by auto
          qed
        next
          show ‹function({⟨0, a⟩})›
            by (rule lsinglfun)
        qed
      qed
    qed
    show ‹{⟨0, a⟩} ` 0 = a ∧ satpc({⟨0, a⟩}, 0, g)›
    proof
      show ‹{⟨0, a⟩} ` 0 = a›
        by (rule func.singleton-apply)
    next
      show ‹satpc({⟨0, a⟩}, 0, g)›
        by (rule singlsatpc)
    qed
  qed
```

```
qed

lemma zainupcs : ‹⟨0, a⟩ ∈ ⋃pcs(A, a, g)›
proof
  show ‹⟨0, a⟩ ∈ {⟨0, a⟩}›
    by auto
next

  show ‹{⟨0, a⟩} ∈ pcs(A, a, g)›
  proof(unfold pcs-def)
    show ‹{⟨0, a⟩} ∈ {t ∈ Pow(nat × A) . ∃m∈nat. partcomp(A, t, m, a, g)}›
    proof
      show ‹{⟨0, a⟩} ∈ Pow(nat × A)›
      proof(rule PowI, rule equalities.singleton-subsetI)
        show ‹⟨0, a⟩ ∈ nat × A›
        proof
          show ‹0 ∈ nat› by auto
        next
          show ‹a ∈ A› by (rule hyp1)
        qed
      qed
    next
      show ‹∃m∈nat. partcomp(A, {⟨0, a⟩}, m, a, g)›
      proof
        show ‹partcomp(A, {⟨0, a⟩}, 0, a, g)›
          by (rule zerostep)
      next
        show ‹0 ∈ nat› by auto
      qed
    qed
  qed
qed

lemma l2′: ‹0 ∈ domain(⋃pcs(A, a, g))›
proof
  show ‹⟨0, a⟩ ∈ ⋃pcs(A, a, g)›
    by (rule zainupcs)
qed
```

Push an ordered pair to the end of partial computation t and obtain another partial computation.

```
lemma shortlem :
  assumes mnat:‹m∈nat›
  assumes F:‹partcomp(A,t,m,a,g)›
  shows ‹partcomp(A,cons(⟨succ(m), g ‘ <t‘m, m>⟩, t),succ(m),a,g)›
proof(rule partcompE[OF F])
  assume F1:‹t ∈ succ(m) → A›
  assume F2:‹t ‘ 0 = a›
  assume F3:‹satpc(t, m, g)›
```

```
  show ?thesis
  proof
    have ljk:‹cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ∈ (cons(succ(m),succ(m)) → A)›
    proof(rule func.fun-extend3[OF F1])
      show ‹succ(m) ∉ succ(m)›
        by (rule  upair.mem-not-refl)
      have tmA:‹t ` m ∈ A›
        by (rule func.apply-funtype[OF F1], auto)
      show ‹g ` ⟨t ` m, m⟩ ∈ A›
        by(rule func.apply-funtype[OF hyp2], auto, rule tmA, rule mnat)
    qed
    have ‹cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ∈ (cons(succ(m),succ(m)) → A)›
      by (rule ljk)
    then have ‹cons(⟨cons(m, m), g ` ⟨t ` m, m⟩⟩, t) ∈ cons(cons(m, m), cons(m,
m)) → A›
      by (unfold succ-def)
    then show ‹cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ∈ succ(succ(m)) → A›
      by (unfold succ-def, assumption)
    show ‹cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ` 0 = a›
    proof(rule trans, rule func.fun-extend-apply[OF F1])
      show ‹succ(m) ∉ succ(m)› by (rule  upair.mem-not-refl)
      show ‹(if 0 = succ(m) then g ` ⟨t ` m, m⟩ else t ` 0) = a›
        by(rule trans, rule upair.if-not-P, auto, rule F2)
    qed
    show ‹satpc(cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t), succ(m), g)›
    proof(unfold satpc-def, rule ballI)
      fix n
      assume Q:‹n ∈ succ(m)›
      show ‹cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ` succ(n)
= g ` ⟨cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ` n, n⟩›
     proof(rule trans, rule func.fun-extend-apply[OF F1], rule upair.mem-not-refl)
        show ‹(if succ(n) = succ(m) then g ` ⟨t ` m, m⟩ else t ` succ(n)) =
    g ` ⟨cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ` n, n⟩›
        proof(rule upair.succE[OF Q])
          assume Y:‹n=m›
          show ‹(if succ(n) = succ(m) then g ` ⟨t ` m, m⟩ else t ` succ(n)) =
    g ` ⟨cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ` n, n⟩›
          proof(rule trans, rule upair.if-P)
            from Y show ‹succ(n) = succ(m)› by auto
          next
            have L1:‹t ` m = cons(⟨succ(m), g ` ⟨t ` m, m⟩⟩, t) ` n›
              proof(rule sym, rule trans, rule func.fun-extend-apply[OF F1], rule
upair.mem-not-refl)
              show ‹ (if n = succ(m) then g ` ⟨t ` m, m⟩ else t ` n) = t ` m›
              proof(rule trans, rule upair.if-not-P)
                from Y show ‹t ` n = t ` m› by auto
                show ‹n ≠ succ(m)›
                proof(rule not-sym)
                  show ‹succ(m) ≠ n›
```

```
                by(rule subst, rule sym, rule Y, rule upair.succ-neq-self)
              qed
            qed
          qed
          from Y
          have L2:‹m = n›
            by auto
          have L:‹ ⟨t ‘ m, m⟩ = ⟨cons(⟨succ(m), g ‘ ⟨t ‘ m, m⟩⟩, t) ‘ n, n⟩›
            by(rule subst-context2[OF L1 L2])
          show ‹ g ‘ ⟨t ‘ m, m⟩ = g ‘ ⟨cons(⟨succ(m), g ‘ ⟨t ‘ m, m⟩⟩, t) ‘ n, n⟩›
            by(rule subst-context[OF L])
        qed
      next
        assume Y:‹n ∈ m›
        show ‹(if succ(n) = succ(m) then g ‘ ⟨t ‘ m, m⟩ else t ‘ succ(n)) =
              g ‘ ⟨cons(⟨succ(m), g ‘ ⟨t ‘ m, m⟩⟩, t) ‘ n, n⟩›
        proof(rule trans, rule upair.if-not-P)
          show ‹succ(n) ≠ succ(m)›
        by(rule contrapos, rule upair.mem-imp-not-eq, rule Y, rule upair.succ-inject,
assumption)
        next
          have X:‹cons(⟨succ(m), g ‘ ⟨t ‘ m, m⟩⟩, t) ‘ n = t ‘ n›
       proof(rule trans, rule func.fun-extend-apply[OF F1], rule upair.mem-not-refl)
            show ‹(if n = succ(m) then g ‘ ⟨t ‘ m, m⟩ else t ‘ n) = t ‘ n›
            proof(rule upair.if-not-P)
              show ‹n ≠ succ(m)›
              proof(rule contrapos)
                assume q:n=succ(m)
                from q and Y have M:‹succ(m)∈m›
                  by auto
                show ‹m∈m›
                  by(rule Nat.succ-in-naturalD[OF M mnat])
              next
                show ‹m ∉ m› by (rule  upair.mem-not-refl)
              qed
            qed
          qed
          from F3
          have W:‹∀ n∈m. t ‘ succ(n) = g ‘ ⟨t ‘ n, n⟩›
            by (unfold satpc-def)
          have U:‹t ‘ succ(n) = g ‘ ⟨t ‘ n, n⟩›
            by (rule bspec[OF W Y])
          show ‹t ‘ succ(n) = g ‘ ⟨cons(⟨succ(m), g ‘ ⟨t ‘ m, m⟩⟩, t) ‘ n, n⟩›
            by (rule trans, rule U, rule sym, rule subst-context[OF X])
        qed
      qed
    qed
  qed
qed
```

```
qed

lemma l2:‹nat ⊆ domain(⋃pcs(A, a, g))›
proof
  fix x
  assume G:‹x∈nat›
  show ‹x ∈ domain(⋃pcs(A, a, g))›
  proof(rule nat-induct[of x])
    show ‹x∈nat› by (rule G)
  next
    fix x
    assume Q1:‹x∈nat›
    assume Q2:‹x∈domain(⋃pcs(A, a, g))›
    show ‹succ(x)∈domain(⋃pcs(A, a, g))›
    proof(rule domainE[OF Q2])
      fix y
      assume W1:‹⟨x, y⟩ ∈ (⋃pcs(A, a, g))›
      show ‹succ(x)∈domain(⋃pcs(A, a, g))›
      proof(rule UnionE[OF W1])
        fix t
        assume E1:‹⟨x, y⟩ ∈ t›
        assume E2:‹t ∈ pcs(A, a, g)›
        hence E2:‹t∈{t∈Pow(nat∗A). ∃m ∈ nat. partcomp(A,t,m,a,g)}›
          by(unfold pcs-def)
        have E21:‹t∈Pow(nat∗A)›
          by(rule CollectD1[OF E2])
        have E22m:‹∃m∈nat. partcomp(A,t,m,a,g)›
          by(rule CollectD2[OF E2])
        show ‹succ(x)∈domain(⋃pcs(A, a, g))›
        proof(rule bexE[OF E22m])
          fix m
          assume mnat:‹m∈nat›
          assume E22P:‹partcomp(A,t,m,a,g)›
          hence E22:‹((t:succ(m)→A) ∧ (t‘0=a)) ∧ satpc(t,m,g)›
            by(unfold partcomp-def, auto)
          hence E223:‹satpc(t,m,g)› by auto
          hence E223:‹∀n ∈ m . t‘succ(n) = g ‘ <t‘n, n>›
            by(unfold satpc-def, auto)
          from E22 have E221:‹(t:succ(m)→A)›
            by auto
          from E221 have domt:‹domain(t) = succ(m)›
            by (rule func.domain-of-fun)
          from E1 have xind:‹x ∈ domain(t)›
            by (rule equalities.domainI)
          from xind and domt have xinsm:‹x ∈ succ(m)›
            by auto
          show ‹succ(x)∈domain(⋃pcs(A, a, g))›
          proof
```

```
show ‹ ⟨succ(x), g ‘ <t‘x, x>⟩ ∈ (⋃pcs(A, a, g))›
proof

  show ‹cons(⟨succ(x), g ‘ <t‘x, x>⟩, t) ∈ pcs(A, a, g)›
  proof(unfold pcs-def, rule CollectI)
    from E21
    have L1:‹t ⊆ nat × A›
      by auto
    from Q1 have J1:‹succ(x)∈nat›
      by auto
    have txA: ‹t ‘ x ∈ A›
      by (rule func.apply-type[OF E221 xinsm])
    from txA and Q1 have txx:‹⟨t ‘ x, x⟩ ∈ A × nat›
      by auto
    have secp: ‹g ‘ ⟨t ‘ x, x⟩ ∈ A›
      by(rule func.apply-type[OF hyp2 txx])
    from J1 and secp
    have L2:‹⟨succ(x),g ‘ ⟨t ‘ x, x⟩⟩ ∈ nat × A›
      by auto
    show ‹ cons(⟨succ(x),g ‘ ⟨t ‘ x, x⟩⟩,t) ∈ Pow(nat × A)›
    proof(rule PowI)
      show ‹ cons(⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩, t) ⊆ nat × A›
      proof
        show ‹⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩ ∈ nat × A ∧ t ⊆ nat × A›
          by (rule conjI[OF L2 L1])
      qed
    qed
  next
   show ‹∃m ∈ nat. partcomp(A, cons(⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩, t), m, a,
g)›
    proof(rule succE[OF xinsm])
      assume xeqm:‹x=m›
      show ‹∃m ∈ nat. partcomp(A, cons(⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩, t), m,
a, g)›
      proof
       show ‹partcomp(A, cons(⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩, t), succ(x), a, g)›
        proof(rule shortlem[OF Q1])
          show ‹partcomp(A, t, x, a, g)›
          proof(rule subst[of m x], rule sym, rule xeqm)
            show ‹partcomp(A, t, m, a, g)›
              by (rule E22P)
          qed
        qed
      next
        from Q1 show ‹succ(x) ∈ nat› by auto
      qed
    next
      assume xinm:‹x∈m›
      have lmm:‹cons(⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩, t) = t›
```

```
                by (rule addmiddle[OF mnat E22P xinm])
              show ‹∃ m∈nat. partcomp(A, cons(⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩, t), m, a,
g)›
                by(rule subst[of t], rule sym, rule lmm, rule E22m)
            qed
          qed
        next
          show ‹⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩ ∈ cons(⟨succ(x), g ‘ ⟨t ‘ x, x⟩⟩, t)›
            by auto
        qed
      qed
    qed
  qed
  qed
  next
    show ‹0 ∈ domain(⋃pcs(A, a, g))›
      by (rule l2′)
  qed
qed

lemma useful : ‹∀ m∈nat. ∃ t. partcomp(A,t,m,a,g)›
proof(rule nat-induct-bound)
  show ‹∃ t. partcomp(A, t, 0, a, g)›
  proof
    show ‹partcomp(A, {⟨0, a⟩}, 0, a, g)›
      by (rule zerostep)
  qed
next
  fix m
  assume mnat:‹m∈nat›
  assume G:‹∃ t. partcomp(A,t,m,a,g)›
  show ‹∃ t. partcomp(A,t,succ(m),a,g)›
  proof(rule exE[OF G])
    fix t
    assume G:‹partcomp(A,t,m,a,g)›
    show ‹∃ t. partcomp(A,t,succ(m),a,g)›
    proof
      show ‹partcomp(A,cons(⟨succ(m), g ‘ <t‘m, m>⟩, t),succ(m),a,g)›
        by(rule shortlem[OF mnat G])
    qed
  qed
qed

lemma l4 : ‹(⋃pcs(A,a,g)) ∈ nat -> A›
proof(unfold Pi-def)
  show ‹ ⋃pcs(A, a, g) ∈ {f ∈ Pow(nat × A) . nat ⊆ domain(f) ∧ function(f)}›
  proof
    show ‹⋃pcs(A, a, g) ∈ Pow(nat × A)›
    proof
```

```
      show ‹⋃pcs(A, a, g) ⊆ nat × A›
        by (rule l1)
    qed
  next
    show ‹nat ⊆ domain(⋃pcs(A, a, g)) ∧ function(⋃pcs(A, a, g))›
    proof
      show ‹nat ⊆ domain(⋃pcs(A, a, g))›
        by (rule l2)
    next
      show ‹function(⋃pcs(A, a, g))›
        by (rule l3)
    qed
  qed
qed

lemma l5: ‹(⋃pcs(A, a, g)) ‘ 0 = a›
proof(rule func.function-apply-equality)
  show ‹function(⋃pcs(A, a, g))›
    by (rule l3)
next
  show ‹⟨0, a⟩ ∈ ⋃pcs(A, a, g)›
    by (rule zainupcs)
qed

lemma ballE2:
  assumes ‹∀x∈AA. P(x)›
  assumes ‹x∈AA›
  assumes ‹P(x) ==> Q›
  shows Q
  by (rule assms(3), rule bspec, rule assms(1), rule assms(2))
```

Recall that *satpc*(t,α,g) == ∀ n ∈ α . t‘succ(n) = g ‘ <t‘n, n> *partcomp*(A,t,m,a,g) == (t:succ(m)→A) ∧ (t‘0=a) ∧ *satpc*(t,m,g) *pcs*(A,a,g) == {t∈Pow(nat∗A). ∃ m. *partcomp*(A,t,m,a,g)}

```
lemma l6new: ‹satpc(⋃pcs(A, a, g), nat, g)›
proof (unfold satpc-def, rule ballI)
  fix n
  assume nnat:‹n∈nat›
  hence snnat:‹succ(n)∈nat› by auto

  show ‹(⋃pcs(A, a, g)) ‘ succ(n) = g ‘ ⟨(⋃pcs(A, a, g)) ‘ n, n⟩›
  proof(rule ballE2[OF useful snnat], erule exE)
    fix t
    assume Y:‹partcomp(A, t, succ(n), a, g)›
    show ‹(⋃pcs(A, a, g)) ‘ succ(n) = g ‘ ⟨(⋃pcs(A, a, g)) ‘ n, n⟩›
    proof(rule partcompE[OF Y])
      assume Y1:‹t ∈ succ(succ(n)) → A›
      assume Y2:‹t ‘ 0 = a›
      assume Y3:‹satpc(t, succ(n), g)›
```

```
hence Y3:‹∀ x ∈ succ(n) . t‘succ(x) = g ‘ <t‘x, x>›
  by (unfold satpc-def)
hence Y3:‹t‘succ(n) = g ‘ <t‘n, n>›
  by (rule bspec, auto)
have e1:‹(⋃ pcs(A, a, g)) ‘ succ(n) = t ‘ succ(n)›
proof(rule valofunion, rule pcs-lem, rule hyp1)
  show ‹t ∈ pcs(A, a, g)›
  proof(unfold pcs-def, rule CollectI)
    show ‹t ∈ Pow(nat × A)›
      proof(rule tgb)
      show ‹t ∈ succ(succ(n)) → A› by (rule Y1)
    next
      from snnat
      show ‹succ(succ(n)) ∈ nat› by auto
    qed
  next
    show ‹∃ m∈nat. partcomp(A, t, m, a, g)›
      by(rule bexI, rule Y, rule snnat)
  qed
next
  show ‹t ∈ succ(succ(n)) → A› by (rule Y1)
next
  show ‹succ(n) ∈ succ(succ(n))› by auto
next
  show ‹t ‘ succ(n) = t ‘ succ(n)› by (rule refl)
qed
have e2:‹(⋃ pcs(A, a, g)) ‘ n = t ‘ n›
proof(rule valofunion, rule pcs-lem, rule hyp1)
  show ‹t ∈ pcs(A, a, g)›
  proof(unfold pcs-def, rule CollectI)
    show ‹t ∈ Pow(nat × A)›
    proof(rule tgb)
      show ‹t ∈ succ(succ(n)) → A› by (rule Y1)
    next
      from snnat
      show ‹succ(succ(n)) ∈ nat› by auto
    qed
  next
    show ‹∃ m∈nat. partcomp(A, t, m, a, g)›
      by(rule bexI, rule Y, rule snnat)
  qed
next
  show ‹t ∈ succ(succ(n)) → A› by (rule Y1)
next
  show ‹n ∈ succ(succ(n))› by auto
next
  show ‹t ‘ n = t ‘ n› by (rule refl)
qed
have e3:‹g ‘ ⟨(⋃ pcs(A, a, g)) ‘ n, n⟩ = g ‘ ⟨t ‘ n, n⟩›
```

```
      by (rule subst[OF e2], rule refl)
    show ‹(⋃pcs(A, a, g)) ‘ succ(n) = g ‘ ⟨(⋃pcs(A, a, g)) ‘ n, n⟩›
      by (rule trans, rule e1,rule trans, rule Y3, rule sym, rule e3)
  qed
 qed
qed
```

7 Recursion theorem

```
theorem recursionthm:
 shows ‹∃!f. ((f ∈ (nat→A)) ∧ ((f‘0) = a) ∧ satpc(f,nat,g))›

proof
 show ‹∃f. f ∈ nat -> A ∧ f ‘ 0 = a ∧ satpc(f, nat, g)›
 proof
  show ‹(⋃pcs(A,a,g)) ∈ nat -> A ∧ (⋃pcs(A,a,g)) ‘ 0 = a ∧ satpc(⋃pcs(A,a,g),
nat, g)›
   proof
    show ‹⋃pcs(A, a, g) ∈ nat -> A›
      by (rule l4)
   next
    show ‹(⋃pcs(A, a, g)) ‘ 0 = a ∧ satpc(⋃pcs(A, a, g), nat, g)›
    proof
      show ‹(⋃pcs(A, a, g)) ‘ 0 = a›
        by (rule l5)
    next
      show ‹satpc(⋃pcs(A, a, g), nat, g)›
        by (rule l6new)
    qed
   qed
 qed
next
 show ‹⋀f y. f ∈ nat -> A ∧
         f ‘ 0 = a ∧
         satpc(f, nat, g) ⟹
         y ∈ nat -> A ∧
         y ‘ 0 = a ∧
         satpc(y, nat, g) ⟹
         f = y›
   by (rule recuniq)
qed

end
```

8 Lemmas for addition

Let's define function t(x) = (a+x). Firstly we need to define a function *g*:*nat* × *nat* → *nat*, such that *g*‘⟨*t*‘*n*, *n*⟩ = *t*‘*succ*(*n*) = *a* + (*n* + *1*) = (*a*

$+ n) + 1 = (t`n) + 1$ So $g`\langle a, b\rangle = a + 1$ and $g(p) = succ(pr1(p))$ and $satpc(t,\alpha,g) \Longleftrightarrow \forall n \in \alpha . t`succ(n) = succ(t`n)$.

```
definition addg :: ‹i›
  where addg-def : ‹addg == λx∈(nat∗nat). succ(fst(x))›

lemma addgfun: ‹function(addg)›
  by (unfold addg-def, rule func.function-lam)

lemma addgsubpow : ‹addg ∈ Pow((nat × nat) × nat)›
proof (unfold addg-def, rule subsetD)
  show ‹(λx∈nat × nat. succ(fst(x))) ∈ nat × nat → nat›
  proof(rule func.lam-type)
    fix x
    assume ‹x∈nat × nat›
    hence ‹fst(x)∈nat› by auto
    thus ‹succ(fst(x)) ∈ nat› by auto
  qed
next
  show ‹nat × nat → nat ⊆ Pow((nat × nat) × nat)›
    by (rule pisubsig)
qed

lemma addgdom : ‹nat × nat ⊆ domain(addg)›
proof(unfold addg-def)
  have e:‹domain(λx∈nat × nat. succ(fst(x))) = nat × nat›
    by (rule domain-lam)
  show ‹nat × nat ⊆
    domain(λx∈nat × nat. succ(fst(x)))›
    by (rule subst, rule sym, rule e, auto)
qed

lemma plussucc:
  assumes F:‹f ∈ (nat→nat)›
  assumes H:‹satpc(f,nat,addg)›
  shows ‹∀ n ∈ nat . f`succ(n) = succ(f`n)›
proof
  fix n
  assume J:‹n∈nat›
  from H
  have H:‹∀ n ∈ nat . f`succ(n) = (λx∈(nat∗nat). succ(fst(x)))` <f`n, n>›
    by (unfold satpc-def, unfold addg-def)
  have H:‹f`succ(n) = (λx∈(nat∗nat). succ(fst(x)))` <f`n, n>›
    by (rule bspec[OF H J])
  have Q:‹(λx∈(nat∗nat). succ(fst(x)))` <f`n, n> = succ(fst(<f`n, n>))›
  proof(rule func.beta)
    show ‹⟨f ` n, n⟩ ∈ nat × nat›
    proof
      show ‹f ` n ∈ nat›
        by (rule func.apply-funtype[OF F J])
```

```
      show ‹n ∈ nat›
        by (rule J)
    qed
  qed
  have HQ:‹f‘succ(n) = succ(fst(<f‘n, n>))›
    by (rule trans[OF H Q])
  have K:‹fst(<f‘n, n>) = f‘n›
    by auto
  hence K:‹succ(fst(<f‘n, n>)) = succ(f‘n)›
    by (rule subst-context)
  show ‹f‘succ(n) = succ(f‘n)›
    by (rule trans[OF HQ K])
qed
```

9 Definition of addition

Theorem that addition of natural numbers exists and unique in some sense. Due to theorem 'plussucc' the term *satpc*(*f*,*nat*,*addg*) can be replaced here with ∀ *n* ∈ *nat* . *f*‘*succ*(*n*) = *succ*(*f*‘*n*).

```
theorem addition:
  assumes ‹a∈nat›
  shows
 ‹∃!f. ((f ∈ (nat→nat)) ∧ ((f‘0) = a) ∧ satpc(f,nat,addg))›
proof(rule recthm.recursionthm, unfold recthm-def)
  show ‹a ∈ nat ∧ addg ∈ nat × nat → nat›
  proof
    show ‹a∈nat› by (rule assms(1))
  next
    show ‹addg ∈ nat × nat → nat›
    proof(unfold Pi-def, rule CollectI)
      show ‹addg ∈ Pow((nat × nat) × nat)›
        by (rule addgsubpow)
    next
      have A2: ‹nat × nat ⊆ domain(addg)›
        by(rule addgdom)
      have A3: ‹function(addg)›
        by (rule addgfun)
      show ‹nat × nat ⊆ domain(addg) ∧ function(addg)›
        by(rule conjI[OF A2 A3])
    qed
  qed
qed

end
```